Verifying Web Browser Extensions' Compliance with Private-Browsing Mode[*]

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

Brown University

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. This feature is in direct tension with support for *extensions*, which can silently void this guarantee.
We create a static 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.
We have retrofitted type annotations to Firefox's APIs and to a sample of actively used Firefox extensions. We used the type system to verify several extensions as safe, find actual bugs in several others (most of which have been confirmed by their authors), and find dubious behavior in the rest. Firefox 20, released April 2, 2013, implements a finer-grained private browsing mode; we sketch both the new challenges in this implementation and how our approach can handle them.

1 Introduction

Modern web browsers are feature-rich systems, providing a highly 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.

[*] This work is partially supported by the US National Science Foundation.

[1] These are distinct from *plugins* such as Flash or Java; we exclude plugins here.

The extension community is vibrant, with over 60 million daily extension users, and billions of total installations [17]. The potential privacy harm from faulty extensions is correspondingly huge. 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. 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 [35, 36]. It is also possible for violations—sometimes flagrant ones [24]—to slip through, and *our work finds more violations, even in extensions that have passed review and been given a security check*. Moreover, if policies ever changed, Mozilla would face the daunting task of re-auditing thousands of extensions.

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 4)
– We implement our approach for the extension APIs found in Mozilla Firefox. Ascribing types to Firefox's APIs is non-trivial; the types must match their quirky idioms with high fidelity in order to be useful. (Section 5)
– We evaluate our system by retrofitting type annotations onto 12 real-world extensions. 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 6)

Due to page limits, we necessarily elide detailed examples of our approach; full details can be found in the companion technical report [21].

2 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 [19], 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 Firefox and Chrome.

2.1 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 various available object types, mapped to factories for constructing instances. As of version 13, Firefox defines 847 contract IDs and 1465 interfaces: a huge API surface.

One of these APIs, `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 5).

2.2 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 [12]. 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 3.2 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.

3 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 impact the design.

3.1 Usability Trade-Offs in Specifying Private Browsing Mode

Private browsing mode is often described as the browser "not remembering anything" about the current session. One implementation approach might be to prohibit all disk writes altogether. Indeed, the initial planning for Firefox's private browsing mode [23] 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."[2]

However, the high-level intent of private-browsing mode 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 the browser to "work like normal" while browsing, and "forget" everything about the session afterward. Such a private-browsing implementation might well handle many forms of persistent state during the session on behalf of the user, such as history, cookies, or cache. Additionally, a user can ask the browser explicitly to take certain stateful actions, such as downloading a file or updating bookmarks. Therefore, simply turning off persistence APIs is not an option.

3.2 Mode Concurrency and Switching

How isolated is private browsing? Chrome (and now Firefox; see Section 8) allows users to have both "normal" and "incognito" windows open concurrently; can this concurrency be exploited to leak data from private browsing? Similarly, can an extension hoard data in private browsing mode, and then release it when the window switches to normal mode?

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 Firefox windows run in the context of each window.[3] When earlier versions of Firefox transition between modes, they close all currently-open private windows and re-open 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.

[2] Even these precautions may not suffice: virtual memory might swap private browsing mode information to persistent storage [4].

[3] The Firefox expert might know about "backstage pass" contexts, which can persist. Such contexts are internal to Gecko and to our knowledge cannot be intentionally accessed by script. Even if they could, we can reflect this in our typed APIs.

In Chrome, extensions are partitioned into two pieces: a background task running in a standalone context, and content scripts running in the context of individual pages. The background task can execute privileged operations, but cannot obtain data about the user's open pages directly. Instead, the content script must send such data to the background task over a well-specified API, and again we can detect and flag such uses.

In short, browsers are 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 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. (We describe how our approach may adapt to Firefox's new design in Section 8.)

4 Our Approach: Type-Based Extension Verification

We assume that the browser vendor has correctly implemented private-browsing mode and focus on whether extensions violate it. We perform this analysis through a *type system* for `JS`. In particular, any accesses to potentially harmful APIs must be syntactically marked in the code, making the reviewers' job a simple search, rather than a reading of the entire extension. Furthermore, we define an *affirmative privacy burden*: rather than require all developers to annotate all code, we require annotations only where code might violate private browsing expectations. Our type system is based on TeJaS, a type system for `JS` [15], with several variations.

4.1 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).
```

In response, the programmer can: 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, which will eliminate the error report but effectively flag the use for audit. Often, very simple changes will suffice as proof:

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

(We show in Section 4.4 how to implement `WeAreNotInPrivateBrowsingMode()`.)

4.2 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 TeJaS’s own type-safety theorem [15], the soundness of Progressive Types [30], 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.

In the following subsections, we explain how the typechecker recognizes the example above as safe, and make precise what “cheating” is and why it is sometimes necessary.

4.3 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 inflexiblly prohibits developers from naming their functions with these banned words. It also proscribes 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).

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.

4.4 The Private-Browsing Type System

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. On top of this, we add support for (unordered) union and (ordered) intersection types. In particular, the type `Bool` is the union of singleton types `(True + False)`.

Safe Types We define a (slightly-simplified) *extension type* that includes all possible `JS` values [32]:

```
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, or other base types, or mutable objects whose fields, if present, are also of this type. The `__proto__` field is a read-only object (about whose fields we know nothing), while `__code__` (when present) models `JS` functions, which are objects with an internal code pointer.

`Ext` is the default type for all expressions, and any `Ext`-typed code need not be annotated, justifying our "lightweight" claims. As Section 4.3 mentioned, developers are free to add more precise types to their code gradually. Any such types

[4] Type *inference* for objects is of little help, as it is often undecidable [26].

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.

Marking APIs with the `@Unsafe` Type We define a new primitive type `@Unsafe` that is ascribed 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 type errors: 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`.

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` values. Recall the revised example from Section 4.1: code that has checked that it is *not* in private-browsing mode may use `@Unsafe` values.

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

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

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 4.1 to typecheck without error.

"Cheating" As pointed out in Section 3.1, 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 (and orthogonal) one we deliberately avoid. Moreover, the control flow of browsers is quite complex [20], 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 real-world `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, 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.

5 Theory to Practice in Firefox

We have laid out our technique that developers would ideally use from the outset as they develop new extensions, and the theorem in Section 4.1 ensures that their efforts would be worthwhile. In this section, we explain the details of instantiating our system for Firefox's APIs. The companion technical report [21] contains additional details and worked examples.

5.1 Translating Typed Interfaces

Most of Mozilla's APIs are defined in typed interface files (written in a variant of WebIDL[5]), 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

[5] http://www.w3.org/TR/WebIDL/

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.

5.2 Encoding `@Unsafe` Values and the Flag for Private Browsing Mode

We define two more IDL attributes, `[Unsafe]` and `[PrivateBrowsingCheck]`, and use them to annotate the relevant properties in the Mozilla environment. Per Section 4, these are translated to the types `@Unsafe` and `True`, respectively.

As mentioned in Section 2.1, 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 benign uses of this API must now be cheated, and their use justified in a security review.

5.3 Encoding the `Components` Object

All of Mozilla's APIs are accessed via roughly this idiom:

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

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" [27], and is encoded in our system with intersection types.[6]

6 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.

[6] Firefox 3's new API, `Component.utils.import("script.js", [obj])`, is not amenable to similar static encoding and consequently requires manual audits.

Extension	Size	Violates PBM?	Confirmed by author?	Violates privacy?
The Middle Mouse Button v1.0	51	No		
Print v0.3.4	59	No		
Rapidfire v0.5	119	No		
Commandrun v0.10.0	147	Yes	Yes	Yes
Open As Webfolder v0.28	153	Yes	No	No
CheckFox v0.9.2	188	No		
The Tracktor Amazon Price Tracker v1.0.7	232	No		
Fireclam v0.6.7	437	Yes	No	No
Cert Viewer Plus v1.7	974	Yes	No	Yes
Textarea Cache v0.8.5	1103	No		
ProCon Latte Content Filter v3.3	2015	Yes	Yes	Yes
It's All Text v1.6.3	2623	Yes	Yes	Yes
Total	8101	6	3	4

Fig. 1. Extensions analyzed for private-browsing violations: note that not all private-browsing violations are actual privacy violations. The technical report [21] has more details on the extensions and annotations, and provides more excerpts.

The results are summarized in Fig. 1. We analyzed 6.8K non-comment lines of code (8.1KLOC including comments and our type definitions), and found private-browsing violations in 6 of the 12 extensions—of which only 4 truly violate privacy. Below, we highlight interesting excerpts from some of these extensions.

6.1 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 some 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 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 `Bool`, `String`, and whichever particular object type is eventually used. This can be useful: distinguishing `True` from `False` values lets us elide dead branches of code and thereby check for private browsing mode, and similarly, distinguishing string literals from each other enables support for `JS` objects' first-class field names [29]. Sometimes, however, this is overly-specific. For instance, a truly boolean-valued variable might be initialized to `true` and later modified to `false`; if its type was inferred as `True`, the subsequent assignment would result in a type error! In such cases, we therefore manually annotate the initializers with their more general types.

Third, we replaced the idiomatic field-existence check `if (!foo.bar)` with `if (!("bar" in foo))`, as the typechecker 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.

6.2 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 detecting privacy violations can be. The companion technical report [21, section VI] contains full details of the necessary annotations.

Almost Safe: Textarea Cache [33] 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:

```
textareaCache.inPrivateBrowsing = function () {
  if (this._PBS) return this._PBS.privateBrowsingEnabled;
  else return true; // N.B.: used to be false
};
textareaCache.beforeWrite = function(node) {
  if (this.inPrivateBrowsing()) return;
  ...
  this.writeToPref(node);
};
```

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.

Unsafe: ProCon Latte Content Filter [25] This "featured" (i.e., nominated as top-quality[7]) 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, a clear (or possibly deliberate) policy violation. The extension author confirmed that this behavior is a bug: URLs were not intended to persist past private browsing mode.

Unsafe: Commandrun [2] 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

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

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):

```
CommandRunHandler = function() {
  this.run = /*:cheat @Unsafe*/function(command, args){ ... };
  this.isCommandAllowed = function(command, args){ ... };
};
CommandRun = {
  onPageLoad: function(event) {
    var win = event.originalTarget.defaultView.wrappedJSObject;
    win.CommandRun = new CommandRunHandler();
} };
```

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 security wrappers to prevent inadvertent tampering with them, but exposes the actual objects via a `wrappedJSObject` field on the wrappers. 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.

7 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.

7.1 Security-Related Efforts

Many recent projects relate to extension security, authoring, or analysis. Several entail pervasive changes within the browser [6, 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.

ADsafety The closest relative of our work is ADsafety [28], which uses TeJaS [15] to verify the correctness of ADsafe [3]. 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! The extension-safety problem here is more refined, and permits such unsafe values in non-private execution.

IBEX Guha et al. [14] 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.

VEX Bandhakavi et al. [5] 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. Additionally, they do not address the conditional safety of API usage which is the hallmark of the private-browsing mode problem.

Beacon Karim et al. [18] design an analysis for Mozilla Jetpack extensions (see Section 2.2) 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.

7.2 Language-Level Analyses

Progressive Types As mentioned, our type system is based on that of Guha et al. [15], with enhancements that simplify reasoning about our relaxed progress guarantees. These enhancements are a form of *progressive typing* [30], 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.

Type Systems for JS TeJaS is one of a handful of type disciplines for JS. The two most fully-featured are the Closure compiler [13] 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.

Language-Based Security Schneider et al. [31] 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 [3, 22, 34] 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.

Certified Browsers Jang et al. [16] present an implementation of a browser kernel implemented in Coq, 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.

8 Breaking News: It Gets Worse!

Firefox 20—released on April 2, 2013—has adopted per-window private browsing granularity (à la Chrome). Unfortunately, existing Firefox APIs enable extensions to access *all* windows, which now include both public and private ones; we have confirmed that this allows sensitive data to leak. Moreover, one such API is used over 6,400 times in our corpus: we expect that extensions using this API—even those using it *safely* in earlier Firefox versions—may now inadvertently violate privacy. We have contacted Mozilla, who indicate that closing this leak (and others) may not be technically feasible.

However, we believe our approach still works. Instead of ignoring non-private-browing code, we must analyze it. We can define another type environment in which `inPrivateBrowsing` is now `False` and a different set of APIs (e.g., window enumeration) are marked either as `@Unsafe` or as returning potentially-`@Unsafe` data. Running the type checker in this environment will then flag potential leakage of private data to public scope.

Bibliography

[1] USENIX Security Symposium. USENIX Association, Berkeley, CA, USA (Aug 2010)

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

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

[4] Aggrawal, G., Bursztein, E., Jackson, C., Boneh, D.: An analysis of private browsing modes in modern browsers. In: [1]
[5] Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M.: VEX: vetting browser extensions for security vulnerabilities. In: [1], pp. 22–22
[6] Barth, A., Felt, A.P., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: Network and Distributed System Security Symposium (NDSS) (Mar 2010)
[7] Cappos, J., Dadgar, A., Rasley, J., Samuel, J., Beschastnikh, I., Barsan, C., Krishnamurthy, A., Anderson, T.: Retaining sandbox containment despite bugs in privileged memory-safe code. In: ACM Conference on Computer and Communications Security (CCS). pp. 212–223. ACM, New York, NY, USA (Oct 2010)
[8] Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 587–606. ACM, New York, NY, USA (2012)
[9] Dhawan, M., Ganapathy, V.: Analyzing information flow in JavaScript-based browser extensions. In: Annual Computer Security Applications Conference (ACSAC). pp. 382–391. IEEE Computer Society, Washington, DC, USA (Dec 2009)
[10] Djeric, V., Goel, A.: Securing script-based extensibility in web browsers. In: [1]
[11] Fredrikson, M., Livshits, B.: RePriv: Re-envisioning in-browser privacy. Tech. rep., Microsoft Research (Aug 2010)
[12] Google. chrome.* APIs – Google Chrome extensions. Retrieved Apr. 2012. http://code.google.com/chrome/extensions/api_index.html
[13] Google. Closure tools — Google developers. Retrieved Nov. 2012. https://developers.google.com/closure/compiler/
[14] Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: IEEE Symposium on Security and Privacy (Oakland). pp. 115–130. IEEE Computer Society, Washington, DC, USA (May 2011)
[15] Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: European Symposium on Programming Languages and Systems (ESOP). pp. 256–275. Springer-Verlag, Saarbrücken, Germany (2011)
[16] Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: USENIX Security Symposium. USENIX Association, Berkeley, CA, USA (Aug 2012)
[17] Jostedt, E. Firefox add-ons cross more than 3 billion downloads! Written Jul. 2012. https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/
[18] Karim, R., Dhawan, M., Ganapathy, V., Shan, C.: An analysis of the Mozilla Jetpack extension framework. In: European Conference on Object-Oriented Programming (ECOOP). pp. 333–355. Springer-Verlag, Beijing, China (2012)

[19] Lerner, B.S.: Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design. Ph.D. thesis, University of Washington Computer Science & Engineering (Aug 2011)

[20] Lerner, B.S., Carroll, M.J., Kimmel, D.P., de la Vallee, H.Q., Krishnamurthi, S.: Modeling and reasoning about DOM events. In: USENIX Conference on Web Application Development (WebApps). USENIX Association, Berkeley, CA, USA (Jun 2012)

[21] Lerner, B.S., Elberty, L., Poole, N., Krishnamurthi, S.: Verifying web browser extensions' compliance with private-browsing mode. Tech. Rep. CS-13-02, Brown University (Mar 2013)

[22] Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript. Tech. rep., Google Inc. (2008)

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

[24] Newton, S. 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/

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

[26] Pierce, B.C.: Bounded quantification is undecidable. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 305–315. ACM Press, New York, NY, USA (1992)

[27] Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)

[28] Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: ADsafety: type-based verification of JavaScript sandboxing. In: USENIX Security Symposium. pp. 12–12. USENIX Association, Berkeley, CA, USA (Aug 2011)

[29] Politz, J.G., Guha, A., Krishnamurthi, S.: Semantics and types for objects with first-class member names. In: Workshop on Foundations of Object-Oriented Languages (FOOL) (2012)

[30] Politz, J.G., de la Vallee, H.Q., Krishnamurthi, S.: Progressive types. In: ACM international symposium on New ideas, new paradigms, and reflections on programming and software. pp. 55–66. Onward! '12, ACM, New York, NY, USA (2012)

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

[32] Scott, D.: Lambda calculus: Some models, some philosophy. In: The Kleene Symposium. pp. 223–265 (1980)

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

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

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

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