REPRIV: Re-Envisioning In-Browser Privacy

Matthew Fredrikson
University of Wisconsin

Benjamin Livshits
Microsoft Research

Microsoft Research Technical Report

MSR-TR-2010-116

Microsoft®
Research

Abstract

In this paper, we present REPRIV, a system for managing and controlling the release of private information from the browser. We demonstrate how always-on user interest mining can effectively infer user interests in a real browser. We go on to discuss an extension framework that allows third-party code to extract and disseminate more detailed information, as well as language-based techniques for verifying the absence of privacy leaks in this untrusted code. To demonstrate the effectiveness of our model, we present REPRIV extensions that perform personalization for Netflix, Twitter, Bing, and GetGlue.

We evaluated several aspects of REPRIV in realistic scenarios. We show that REPRIV's default in-browser mining can be done with no noticeable overhead to normal browsing, and that the results it produces converge quickly. We then go on to show similar results for each of our case studies: that REPRIV enables high-quality personalization, as shown by cases studies in news and search result personalization we evaluated on thousands of instances, and that the performance impact each case has on the browser is minimal. We conclude that personalized content and individual privacy on the web are not mutually exclusive.

1. Introduction

The motivation of this work comes from the observation that in today's web there are two distinct groups, *users* and *service providers* such as Amazon, Google, Microsoft, Facebook and the like. Service providers are interested in learning as much about their users as they can so that they can better target their ads or provide content personalization. Users might welcome content, ad, and site personalization as long as it does not compromise their privacy.

In today's web, for service providers, personalization opportunities are limited. Even if sites like Amazon and Facebook allow or sometimes require authentication, service providers only know as much about the user as can be gathered through interaction with the site. A user might only spend a few minutes a day on Amazon.com, for example. This is minuscule compared to the amount of time the same user spends in the browser. This suggests a simple lesson: the browser knows much more about you than any particular site you visit. Based on this observation, we suggest the following strategy, which forms the basis for REPRIV:

1. Let the browser infer information about the user's interests based on his browsing behavior, the sites he visits, his prior history, and detailed interactions on web sites of interest to form a *user interest profile*.
2. Let the browser control the release of this information. For instance, upon the request of a site such as `Amazon.com` or `BarnesAndNoble.com`, the user will be asked for a permission to send her high-level interests to the site. This is similar to prompting for the permission to obtain the geo-location in today's mobile browsers. By default, more explicit information than this, such as the history of visited URLs would not be exposed to the requesting site. It is important that the user stay in control of the information that is released.
3. In addition to default user interest mining, REPRIV allows service providers to register extensions that would perform information extraction within the browser. For instance, a Netflix extension (or *miner*) may extract information pertinent to what movies the user is interested in. The miner may use the history of visiting `Fandango.com` to see what movies you saw in theaters in the past. REPRIV miners are statically verified at the time of submission to disallow undesirable privacy leaks.

This approach is attractive for web service providers because they get access to user's preferences without the need for complex data mining machinery and is in any case based on very limited information. It is also attractive for the user because of better ad targeting and content personalization opportunities. Moreover, this approach opens up an interesting new business model: service providers can incentivize users to release their preferences in exchange for store credit, ad-free browsing, or access to premium content. Compared to prior research [12, 33], the appeal of REPRIV is considerably more extensive as it enables the following broad applications:

1. **Personalized search.** Search results from a variety of search engines can be re-ranked to match user's preferences as well as their browsing history (Section 6.1).
2. **Site personalization.** Sites such as Google News, `CNN.com`, or `overstock.com` can be easily adopted within the browser to match user's news or shopping preferences (Section 6.2).
3. **Ad targeting.** Although we do not explicitly focus on ad personalization in this apper, REPRIV enables client-based ad-targeting as suggested by Adnostic [33] and Privad [].

Note that REPRIV is largely orthogonal to in-private browsing modes supported by modern browsers. While it is still possible for a determined service provider to perform user tracking unless they combile REPRIV with a browser privacy mode, it is our hope that the service provider will opt for explicitly requesting user preferences through the REPRIV protocol rather than using a back door.

1.1 Contributions

Our paper makes the following contributions:

- REPRIV, a system for controlling the release of private information within the browser. We demonstrate how built-in data mining of user interests can work in a real browser called `C3`.
- REPRIV **protocol.** We propose a protocol on top of HTTP that can be used to seamlessly integrate REPRIV with existing web infrastructure. We also show how pluggable extensions can be used to extract more detailed information, and how to check these third-party miners for unwanted privacy leaks.
- REPRIV **miners.** Using four realistic miner examples, we show how custom miners can be developed in REPRIV.
- **Miners verification.** We describe a browser API and a type system based on the Fine programming language [] that ensures that miners installed in the brower cannot result in privacy leaks.
- **Evaluation.** Implementation and evaluation in real-life scenarios. We demonstrate that REPRIV mining can be done with minimal overhead to the end-user latency. We also show the efficacy of REPRIV mining on real-life browsing sessions and conclude that REPRIV is able to learn user preferences quickly and sufficiently for many web personalization tasks. We demonstrate the utility of REPRIV by performing two large-scale case studies, one targeting news personalization, and the other focusing on search result personalization, both evaluated on real user data.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 gives motivation for the specification inference techniques REPRIV uses. Section 3 talks about our implementation. Section 4 discusses custom REPRIV miners. Section 5 describes our experimental evaluation. Section 6 describes two detailed case studies, one focusing on news and the other on search personalization. Section 7 touches upon everything else. Finally, Sections 8 and 9 describe related work and conclude.

2. Overview

We begin with a high-level discussion in Section 2.1 of existing efforts to preserve privacy on the web, and how REPRIV fits into this context. Section 2.2 talks about site personalization and Section 2.3 argues for third-party personalization extensions or "miners".

2.1 Background

One definition of privacy common in popular thought and law is summarized as follows: individual privacy is a person's right to control information about one's self, both in terms of how much information others have access to, and the manner in which others may use it. The web as it currently stands is different from how it was initially concieved; it has transformed from a passive medium to an active one where users take part in shaping the content they receive. One popular form of active content on the web is *personalized* content, wherein a provider uses certain characteristics of a particular user, such as their demographic or previous behaviors, to filter, select, or otherwise modify the content that it ultimately presents. This transition in content raises serious concerns about privacy, as arbitrary personal information may be required to enable personalized content, and a confluence of factors has made it

difficult for users to control where this information ends up, and how it is used.

Because personalized content presents profit opportunity, businesses have incentive to adopt it quickly, oftentimes without user consent. This creates situations that many users percieve as a violation of privacy. A prevalent example of this is already seen with online targeted advertising, such as that offered by Google AdSense [10]. By default, this system tracks users who enable browser cookies across all websites that choose to partner with it. This tracking can be arbitrarily invasive as it pertains to the user's behavior at partner sites, and in most cases the user is not explicitly notified that the content they choose to view also actively tracks their actions, and transmits it to a third party (Google). While most services of this type have an opt-out mechanism that any user can invoke, many users are not even aware that a privacy risk exists, much less that they have the option of mitigating it.

As a response to concerns about individual privacy on the web, developers and researchers continue to release solutions that return various degrees of privacy to the user. One well-known example is the *private browsing modes* available in most modern browsers, which attempt to conceal the user's identity across sessions by blocking access to various types of persistent state in the browser [1]. However, a recent study [1] demonstrated that none of the major browsers implement this mode correctly, leading to alarming inconsistencies between user expectations and the features offered by the browser. Even if private browsing mode were implemented correctly, it inherently poses significant problems for personalized content on the web, as sites are not given access to the information needed to perform personzlization.

Others have attempted to build schemes that preserve the privacy of the user while maintaining the ability to personalize content. Most examples [9] [12] [17] [33] concern targeted advertising, given its prevalence and well-known privacy implications. For example, both PrivAd [12] and Adnostic [33] are end-to-end systems that preserve privacy by performing all behavior tracking on the client, downloading *all* potential advertisements from the advertisor's servers, and selecting the appropriate ad to display locally on the client. Although these systems differ in details regarding accounting and architecture, they share a basic strategy for maintaining user privacy: keep sensitive information local to the user, to simplify the matter of control.

The goal of REPRIV is to enable general personalized content on the web in a privacy-conscious manner. Like PrivAd and Adnostic, REPRIV does this by keeping all of the sensitive information necessary to perform personalization close to the user, within the browser. However, REPRIV differs from these systems both technically and in the notion of privacy it considers. Because REPRIV does not target a specific application, it does not attempt to completely hide all personal information from the party responsible for providing personalized content. Aside from the improbable technical advances needed to make such a system practical, it is not clear that content providers would take part in such a scheme, as they would loose access to the valuable user data that they currently use to improve their products and increase efficiency. Rather, REPRIV leaves it to the user to decide which parties may access the various types of data stored inside the browser, and manages dissemination accordingly in a secure manner.

We posit that expecting the user to make this decision is not only reasonable, but necessary given the constraints discussed above. The basis of this decision must be two-fold, depending both on the trust the user has in the content provider, as well as the incentive the content provider gives the user for access to his data. However, this type of decision is ultimately similar to the type of decision a user makes when signing up for an account at amazon.com or netflix.com: if he agrees to the terms in the privacy policy, then he has deemed the benefit offered by that site worth the reduction in personal privacy needed to obtain it. This is the same negotiation that REPRIV relies on to protect user privacy while still enabling a diverse set of personalized applications. Thus, the challenge of REPRIV is to facilitate the collection of personal information from the browser in a manner flexible enough to enable existing and future personalized applications, while maintaining explicit user control over how that information is used and disseminated to third parties on the web.

2.2 Motivating Personalization Scenarios

In designing the core behavior mining mechanisms in REPRIV, we kept several applications in mind to guide our considerations. Our goal with core behavior mining is to remain as general and flexible as possible, while allowing the user precise, total control over the information about them that is released to remote parties.

Content Targeting: Commonplace on many online merchant websites is content targeting: the inference and strategic placement of content likely to compel the user, based on previous behavior. Although popular sites such as amazon.com and netflix.com already support this functionality without issue, the amount of personal information collected and maintained by these sites have real implications for personal privacy [24] that may surprise many users. Additionally, the fact that the personal data needed to implement this functionality is vaulted on a particular site is an inconvenience for the user, who would ideally like to use their personal information to recieve a better experience on a competitor's site. By keeping all of the information needed for this application in the browser, REPRIV can solve both problems.

As a concrete example, consider that news sites should be able to target specific stories to users based on their interests. This could be done in a hierarchical fashion, with various degrees of specificity. For example, when a user navigates `nytimes.com`, the main site could present the user with easy access to relevant types of stories (e.g. technology, politics, ...). When the user navigates to more specific portions of the site, as in looking only at articles related to technology, then the site should be able to query for specific interest levels on subtopics, to prioritize stories that best match the user. As the site attempts to provide this functionality, the user should be able to decline requests for personal information, and possibly offer related personal information that is not as specific or personally-identifying as a more private alternative. Notice that nytimes.com does not play a special role in this process; immediately after visiting nytimes.com, a competing site such as reuters.com could utilize the same information about the user to provide a similar personalized experience.

Targeted Advertising: Advertising serves as one of the primary enablers of free content on the web, and *targeted advertising* allows merchants to maximize the efficiency of their efforts. REPRIV should facilitate this task in the most direct way possible by allowing advertisers to consult the user's personal information, without removing consent from the picture. Advertisers have incentive to use the accurate data stored by REPRIV, rather than collecting their own data, as the browser-computed interests are more representative of the user's complete browsing behavior. Additionally, consumers are likely to select businesses who engage in practices that do not seem invasive.

Most targeted advertisement schemes today make use of an *interest taxonomy*, that characterize market segments in which a user is most likely to show interest. Thus, for REPRIV to properly facilitate existing targeted advertising schemes, it must allow a third-party to infer this type of information with explicit consent from the user.

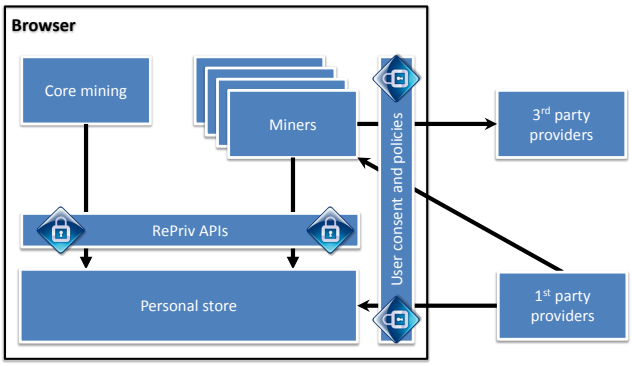


Figure 1: REPRIV architecture

2.3 Personalization Extensions

While the core mining mechanism in REPRIV is meant to be as general-purpose as possible, the pace at which new personalized web applications is appearing suggests that REPRIV will need an extra degree of flexibility to support up-and-coming apps. A large part of our work focuses on an extension platform that enables near-arbitrary programmatic interaction with the user's personal data, in a verifiably privacy-preserving manner. Due to the technicial difficulty of achieving this goal, we focus our efforts on a few areas described below.

Topic-Specific Functionality: Users may spend a disproportionate amount of time at particular types of sites, e.g. movie-related, science, or finance sites. These users are likely to expect a more specific degree of personalization on these sites than the general-purpose core mining can provide. To facilitate this, third-party authors should be able to write extensions that have specific understanding of user interaction with these websites, and are able to mediate REPRIV's stored user information accordingly. For example, a plugin should be able to track the user's interaction with Netflix, observe which movies he likes and dislikes, and update his interest profile to reflect these preferences. Another example arises with search engines: a single extension should be able to interpret interactions with all popular search engines, perform an analysis to determine which interest categories the observed search queries relate to, and update the user's profile accordingly.

Web Service Relay: A popular trend on the web is to open proprietary functionality to independent developers through API's over HTTP. Many of these API's have direct implications for personalization. For example, Netflix now has an API that allows a third-party developer to programmatically access information about the user's account, including their movie preferences and purchase history. Other examples allow a third-party developer to submit portions of a user's overall preference profile or history to recieve content recommendations or hypothesized ratings; getglue.com, hunch.com, and tastekid.com are all examples of this. REPRIV extensions should be able to act as intermediaries between the user's personal data and the services offered by these API's. For example, when a user navigates to fandango.com, the site can query an extension that in turn consults the user's Netflix interactions and amazon.com purchases, and returns useful derived information to fandango.com for personalized showtimes or film reviews.

Direct Personalization: In many cases, it is not reasonable to expect a website to keep up with the user's expectations when it comes to personalization. It may be simpler and more direct to write an extension that can access REPRIV's repository of user information, and modify the presentation of selected sites to implement a degree of personalization that the site is unwilling to provide. To facilitate this need, REPRIV extensions should be able to interact with and modify the DOM structure of selected websites to reflect the contents of the user's personal information. For example, REPRIV should allow an extension that activates only when the user visits `nytimes.com`, and reconfigures the layout of the stories that are presented to reflect the interest topics that are most prevalent in REPRIV's personal information store.

3. Technical Issues

This section is organized as follows. Section 3.1 discussed browser modifications we implemented to support REPRIV. Section 3.2 discusses support for REPRIV miners.

3.1 Browser Modifications

Our current research prototype of REPRIV is built on top of `C3`, a research browser developed in .NET. However, we believe that other browsers can be modified in a very similar manner. We modified `C3` in the following ways to add support for REPRIV:

- Added a behavior mining algorithm that observes users' browsing behavior and automatically updates a profile of user interests (Section 3.1.1).
- Implemented a communication protocol that sits on top of HTTP and allows web sites to utilize the information maintained by REPRIV in the browser (Section 3.1.2).
- Exposed an API that allows third-party extensions to utilize the information maintained by REPRIV, and interact programatically with web sites (Section 3.2).

3.1.1 User Behavior Mining

The goal of our general-purpose behavior mining algorithm is to provide relevant parties with two types of information about the user:

- Top-n topics of interest, where n can vary to suit the needs of each particular application,
- The level of interest in a given set of topics, normalized to a reasonable scale.

Our approach works by classifying individual documents viewed in the browser, and keeping related aggregate information of total browsing history in the personal store.

Interest Categories: To characterize user interests, we use a hierarchical taxonomy of document topics maintained by the Open Directory Project [26] (ODP). The ODP classifies a portion of the web according to a hierarchical taxonomy with several thousand topics, with specificity increasing towards the leaf nodes of the corresponding tree. We use only the most general two levels of the taxonomy, which account for 450 topics. To convey the level of specificity contained in our interest hierarchy, a small portion is presented in Figure 2.

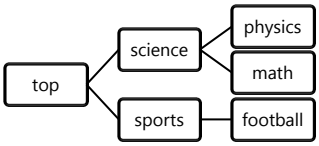


Figure 2: Portion of taxonomy

Classifying Documents: Of primary importance for our document classification scheme is performance: REPRIV's default behavior must not impact normal browsing activities in a noticeable way. This immediately rules out certain solutions, such as querying existing web API's that provide classification services. We selected the Naïve Bayes classification algorithm for its well-known performance in document classification tasks, as well as its low computation cost on most problem instances.

To create our Naïve Bayes classifier, we obtained 3,000 documents from each category of the first two levels of the ODP taxonomy. We selected attribute words as those that occur in at least 15% of documents for at least one category, not including stop words such as "a", "and", and "the". We then ran standard Naïve Bayes training on the corpus, calculating the needed probabilities $P(w_i \mid C_j)$, for each attribute word w_i and each class C_j. Calculating document topic probabilities at runtime is then reduced to a simple log-likelihood ratio calculation over these probabilities.

To ensure that the cost of running topic classifiers on a document does not impinge on browsing activities, this computation is done in a background worker thread. When a document has finished parsing, its `TextContent` attribute is queried and added to a task queue. When the background thread activates, it consults this queue for unfinished classification work, runs each topic classifier, and updates the personal store. Due to the interactive characteristics of internet browsing, i.e. periods of bursty activity followed by downtime for content consumption, there are likely to be many opportunities for the background thread to complete the needed tasks.

Aggregate Statistics: REPRIV uses the classification information from individual documents to relate aggregate information about user interests to relevant parties. The first type of information that REPRIV provides is the "top-n" statistic, which reflects n taxonomy categories that comprise more of the user's browsing history than the other categories. Computing this statistic is done incrementally, as browsing entries are classified and added to the personal store.

The second type of information provided by REPRIV is the degree of user interest in a given set of interest categories. For each interest category, this is interpreted as the portion of the user's browsing history comprised of sites classified with that category. This statistic is efficiently computed by indexing the database underlying the personal store on the column containing the topic category.

3.1.2 Interest Protocol

REPRIV allows third-party websites to query the browser for two types of information that are computed by default when REPRIV runs. The protocols are depicted graphically in Figure 3. The design of these protocols is constrained by three separate concerns:

1. Secure dissemination of personal information. The user should have explicit control over the information that is passed from the browser to the third-party website. Additionally, the user-driven declassification process should be intuitive and easy to understand: when the user is prompted with a dialog box regarding a request for personal information, it should be obvious what information is at stake, and what measures the user must take to either allow or disallow the dissemination. Finally, it should be possible to communicate this information over a channel secure from eavesdropping.
2. Backwards compatibility with existing protocols. Site operators should not need to run a separate daemon on behalf of REPRIV users. Rather, it should be possible to incorporate the information made available by REPRIV with minor changes to existing software.

We will now walk through each step of the protocol. There are two shown in Figure 3; one for each type of information that can be queried (*top-n* interests and specific interest level by category). However, they differ only in minor ways regarding the types of information communicated.

The client signals its ability to provide personal information by including a `repriv` element in the `Accept` field of the standard HTTP header. If the server daemon is programmed to understand this flag, then it may respond with an HTTP 300 message, providing the client with the option of subsequently requesting the default content, or providing personal information to receive personalized content. The information requested by the server is encoded as URL parameters in one of the content alternatives listed in this message. For example, the server in Figure 3(b) requests the user's interest in the topic "*category-n*", which is encoded by specifing `catN` as the value for the `interest` variable. At this point, the browser prompts the user regarding the server's information request, in order to declassify the otherwise prohibited flow from the personal store to an untrusted party. If the user agrees to the information release, then the client responds with a `POST` message to the originally-requested document, which additionally contains the answer to the server's request. Otherwise, the connection is dropped.

3.2 Miner Support

To support a degree of flexibility and allow future personalization applications to integrate into its framework, REPRIV provides a mechanism for loading third-party software that utilizes the personal store. We call REPRIV extensions *Miners*, to reflect the fact that they are intended to assist with novel behavior mining tasks. Of paramount importance to supporting miners correctly is ensuring that *(1)* they do not leak private user data to third parties without explicit consent from the user, and *(2)* does not compromise the integrity of the browser, including other miners. The majority of our technical discussion regarding miners addresses these concerns.

3.2.1 Security Policies

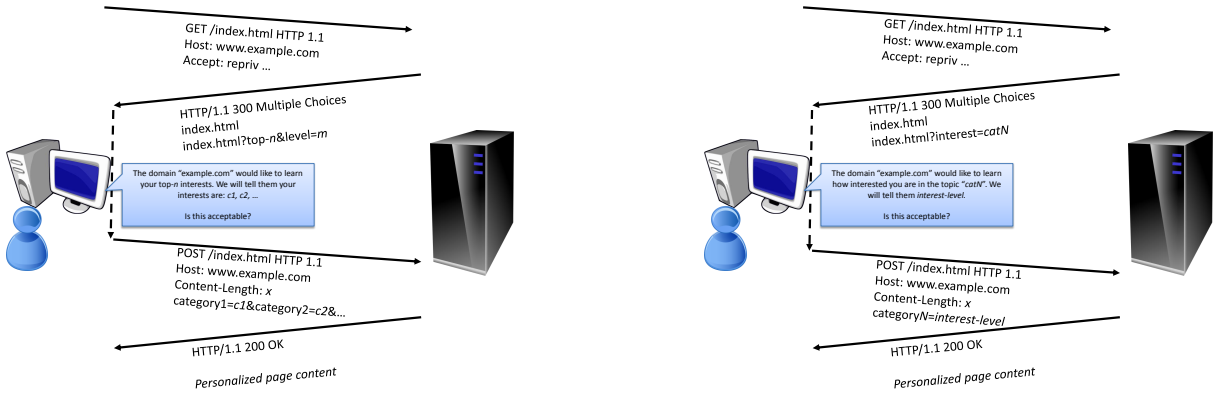
To support a diverse set of extensions while maintaining control over the sensitive information contained in the personal store, REPRIV allows extension authors to express the capabilities of their code in a simple policy language. At the time of installation, users are presented with the extension's list of needed capabilities, and have the option of allowing or disallowing any of the individual capabilities. Several of the policy predicates refer to *provenance labels*, which are $\langle host, extensionid \rangle$ pairs. All sensitive information used by miners is tagged with a set of these labels, which allow policies to reason about information flows involving arbitrary $\langle host, extensionid \rangle$ pairs. REPRIV's policy language is based on the predicates in Figure 4.

Given a list of policy predicates regarding a particular miner, the policy for that extension is interpreted as the conjunction of each predicate in the list. This is equivalent to behavioral whitelisting: unless a behavior is implied by the predicate conjunction, the miner does not have permission to exhibit it. Each miner is associated with one security policy, that is active throughout the lifespan of the miner; it is not possible in general to add or remove predicates from the miner's policy after the miner is written.

3.2.2 Tracking Sensitive Information

When a miner makes a call to REPRIV requesting information from the personal store, special precautions must be taken to ensure that the returned information is not misused. Likewise, when a miner writes information to the store that is derived from content on pages viewed by the user, REPRIV must ensure that the user's wishes are not violated. All REPRIV functionality that returns sensitive information to miners first encapsulates it in a private data type `tracked`, which contains metadata indicating the provenance of that information.

This allows REPRIV to take the provenance of data into account when it is used by miners. Additionally, `tracked` is opaque – it does not allow miner code to directly reference the tracked data that it encapsulates without invoking a REPRIV mechanism that prevents misuse. This means that REPRIV can ensure complete noninterference, to the degree mandated by the miner's policy. Whenever the miner would like to perform a computation over the



(a) *top-n* interests

(b) Interest level by category

Figure 3: Communication protocols for personal information.

encapsulated information, it must call a special `bind` function that takes a function-valued argument and returns a newly-encapsulated result of applying it to the `tracked` value. This scheme prevents leakage of sensitive information, as long as the function passed to `bind` does not cause any side effects. We discuss verification of this property below.

3.2.3 Verifying Miners

Verifying miners against their stated security properties is an entirely static process. This eliminates the need for costly run-time checks, and ensures that a security exception will never interrupt a browsing session. To meet this goal, we require that all untrusted miners be written in Fine [30], a security-typed programming language. Fine allows programmers to express dependent types on function parameters and return values, which forms the basis of REPRIV's verification mechanism.

All REPRIV functionality is exposed to miners through Fine wrappers of API functions. The interface for these wrappers specifies dependent type refinements on key parameters that reflect the consequence of each API function on the relevant policy predicates. Two example interface definitions are given in Figure 5. The first example, `MakeRequest`, is the API used by miners to make HTTP requests; several policy interests are operative in its definition. The second argument of `MakeRequest` is a string that denotes the remote host with which to communicate, and is refined with the formula

```
AllCanCommunicateXHR host p
```

where `p` is the provenance label of the buffer to be transmitted. This refinement ensures that a miner cannot call `MakeRequest` unless its policy includes a `CanCommunicateXHR` predicate for each element in the provenance label `p`. Because the REPRIV API is very limited, we can rest assured that this is the only function that impacts the `CanCommunicateXHR` predicate, giving us a strong argument for correctness of implementation.

Notice as well that the third argument, as well as the return value, of `MakeRequest`, are of the dependent type `tracked`. `tracked` types are indexed both by the type of the data that they encapsulate, as well as the provenance of that data. The third argument is the request string that will be sent to the host specified in the second argument; its provenance plays a part in the refinement on the host string discussed above. The return value has a provenance label that is refined in the fifth argument. The refinement specifies that the provenance of the return value of `MakeRequest` has all elements of the provenance associated with the request string, as well as a new provenance tag corresponding to $\langle$`host`, `eprin`$\rangle$, where `eprin` is the extension principal that invokes the API. The refinement on the fourth argument ensures that the extension passes its actual `ExtensionId` to `MakeRequest`. These considerations ensure that the provenance of information passed to and from `MakeRequest` is available for all necessary policy considertations.

As discussed above, verifying correct enforcement of information flow properties in REPRIV requires checking that functional arguments passed to `bind` are side effect-free. Fortunately, Fine does not provide any default support for creating side effects, as it is purely functional and does not contain facilities for interacting with the operating system. Therefore, the only opportunities for a miner to create a side effect are due to the REPRIV API; our verification task reduces to ensuring that API's which create side effects are not called from code that is invoked by `bind`, as `bind` provides direct access to data encapsulated by `tracked` types.

We use *affine types* [30] to gain this property, as follows. Each API function that may create a side effect takes an argument of affine type `mut_capability` (short for "mutation capability"), which indicates that the caller of the function has the right to create side effects. REPRIV passes each miner a value of type `mut_capability` to its `main` function, which the miner must in turn pass to each location that calls a side-effecting function. Because `mut_capability` is an affine type, and the functional argument of `bind` does not specify an affine type, the Fine type system will not allow any code passed to `bind` to reference a `mut_capability` value, and there is no possibility of creating a side effect in this code. As an example of this construct in the REPRIV API, observe that both API examples in Figure 5 create side effects, so their interface definitions specify arguments of type `mut_capability`.

3.2.4 Verification Philosophy

The policy associated with a miner is expressed at the top of its source file, using a series of Fine `assume` statements: one `assume` for each conjunct in the overall policy. An example of this is shown in Figure 9 on the right hand side, where the policy assumptions of the miner are 3–5 lines of the source code. Given the type refinements on all REPRIV API's, verifying that the miner correctly implements its stated policy is reduced to an instance of Fine type checking. The soundness of this technique rests on three assumptions:

- The soundness of the Fine type system, and the correctness of its implementation. The soundness of the type system was previously demonstrated in a technical report [30].
- The correctness of the dependent type refinements placed on the API functions. This amounts to less than 100 lines of code, which reasons about a relatively simple logic of policy predi-

$CanCaptureEvents(t, \langle h, e\rangle)$	indicates that the extension can capture events of type t on elements tagged $\langle h, e\rangle$.
$CanReadDOMElType(t, h)$	indicates that the extension can read DOM elements of type t from pages hosted by h.
$CanReadDOMElClass(c, h)$	indicates that the extension can read DOM elements of class c from pages hosted by h.
$CanReadDOMId(i, h)$	indicates that extension e can read DOM elements with ID i from pages hosted by h.
$CanWriteDOMElType(t, \langle h_1, e\rangle, h_2)$	indicates that the extension can modify DOM elements of type t with data tagged $\langle h_1, e\rangle$ on pages hosted by h_2.
$CanUpdateStore(d, \langle h, e\rangle)$	indicates that the extension can update the personal store with information tagged $\langle h, e\rangle$.
$CanReadStore(\langle h, e\rangle)$	indicates that the extension can read items in the personal store tagged $\langle h, e\rangle$.
$CanCommunicateXHR(h_1, \langle h_2, e\rangle)$	indicates that the extension can communicate information tagged $\langle h_2, e\rangle$ to host h_1 via XHR-style requests.
$CanServeInformation(h_1, \langle h_2, e\rangle)$	indicates that the extension can serve programmatic requests to sites hosted by h_1, containing information tagged $\langle h_2, e\rangle$. An example of a programmatic request is an invocation of an extension function from the JavaScript on a site in d.
$CanReadLocalFile(f)$	indicates that the extension can read data from the local file f.
$CanHandleSites(h)$	indicates that the extension can set load handlers on sites hosted by h.

Figure 4: Selected security policy predicates. A full listing is available in Appendix C.

```
val MakeRequest:
  p:provs ->
  {host:string | AllCanCommunicateXHR h p} ->
  t:tracked<string,p> ->
  {eprin:string | ExtensionId eprin} ->
  fp:{p:provs | forall (pr:prov).(InProvs pr p) <=>
         (InProvs pr p || pr = (P h eprin))} ->
  mut_capability ->
  tracked<xdoc,fp>

val AddEntry:
  ({p:provs | AllCanUpdateStore p}) ->
  tracked<string,p> ->
  string ->
  tracked<list<string>,p> ->
  mut_capability ->
  unit
```

Figure 5: Example API definitions

cates. Furthermore, because the REPRIV API is very limited, it is simpler to argue that refinements are placed on all necessary arguments to ensure sound enforcement. In other words, the API usually only provides one function for producing a particular type of side effect, so it is not difficult to check that the appropriate refinements are placed at all necessary points.

- The correctness of the underlying browser's implementation of functions provided by the REPRIV API. For REPRIV, we used `C3`, an experimental managed-code browser. `C3` is written in a memory-managed language (C#), providing assurance that it does not contain memory corruption vulnerabilities. The logical correctness of `C3` code needed by REPRIV has not been formally verified, but doing so is a goal of future work.

We stress that these are modest requirements for the trusted computing base, and point towards the overall soundness of REPRIV.

4. REPRIV Miners

In this section, we discuss four example miners that illustrate the concepts discussed in Section 3: TwitterMiner, BingMiner, NetflixMiner, and GlueMiner.

4.1 Miner Patterns

In general, miners can provide a wide range of functionality when it comes to updating the personal store with information that reflects the user's browser-related behaviors. In this section, we present several examples of miners that fall into three patterns of functionality that we envision many potential miners following. The policies for each category can be *templatized*, easing the burden on miner developers who wish to create variations on these basic patterns. The three patterns are summarized in Figure 6.

The first miner pattern, "site-specific parsing", includes extensions that are aware of the layout and semantics of specific websites, and are able to update the user's interest profile accordingly. For example, TwitterMiner invokes REPRIV's document classifier over the text contained in the user's latest tweets, and BingMiner classifies the user's search terms. Miners that follow this pattern either need to send HTTP requests to relevant web API's, as in the case of TwitterMiner, or read the relevant DOM elements from particular sites, as with BingMiner. They invariably require permission to update the personal store with information derived from these sources.

The second pattern, "category-specific information", returns detailed information about the user's interactions with specific types of sites to services that request it via a JavaScript interface. NetflixMiner is an example of this pattern; the user's interactions with pages hosted by `netflix.com` are monitored, and information is added to the personal store to reflect this. When a third-party site, such as `fandango.com`, would like to personalize based on the user's recent movie interests, NetflixMiner queries the store to retrieve the list of most recently-viewed entries by genre, and returns the relevant titles to the third-party site. In addition to the capabilities required by site-specific parsing miners, miners that follow this pattern also need the ability to read from the store, and return tagged information to specific sites via a programmatic interface.

The final pattern, "web service relay", acts as a privacy-conscious intermediary between the user's personal information, and websites that provide useful services using this information. Miners in this category expose functionality via a JavaScript interface, and query a third-party web service with data from the personal store to implement this functionality. For example, GlueMiner returns movies similar to those recently viewed by the user

Pattern	Policy Template
Site-specific parsing	For the domain d of interest, either *CanCommunicateXHR*(d) or *CanReadDOM**(d, _) *CanUpdateStore*(_, d) *CanHandleSites*(d) (optional, depending on the semantics of the miner) *CanCaptureEvents*(_, d) (optional, depending on the semantics of the miner)
Category-specific information	For the domain d of interest, either *CanCommunicateXHR*(d) or *CanReadDOM**(d, _) *CanUpdateStore*(*Tag*(_, d)) *CanHandleSites*(d) (optional, depending on the semantics of the miner) *CanCaptureEvents*(_, d) (optional, depending on the semantics of the miner) *CanReadStore*(*Tag*(_, d)) For each domain p that can request category-specific information, *CanServeInformation*(p, *Tag*(_, d))
Web service relay	For the API provider a and each provenance tag t sent to a *CanCommunicateXHR*(a, t) and *CanReadStore(t)* For each domain p that can make requests, *CanServeInformation(p, t)* and *CanServeInformation(p, a)*

Figure 6: Miner patterns and their policy templates

by reading store entries created by NetflixMiner, sending them to the API provided by `getglue.com`, and returning the results to the JavaScript that requested this information.

4.2 Miner Examples

In this section we present several specific REPRIV extensions for Netflix (Section 4.2.3), `hunch.com` (Section 4.2.4), and Bing (Section 4.2.1). Figure 8 compares the sizes of these miners as well as the time it takes the Fine compiler to verify them.

4.2.1 BingMiner

BingMiner is a simple example of a REPRIV extension that understands the semantics of a particular website, and is able to update the personal store accordingly. The functionality of this miner is straightforward: when the user navigates to a site hosted by `bing.com`, the extension recieves a callback from the browser, at which point it attaches a listener on `submit` events for the search form. Whenever the user sends a search query to Bing, the callback method receives the contents of the query. It then invokes REPRIV's document classifier to determine which categories the query may apply to, and updates the personal store accordingly.

To carry out these tasks, BingMiner needs four capabilities from REPRIV, as depicted in Figure 7.

1. It must be able to listen for DOM `submit` events on sites from `bing.com`.
2. In order to determine which elements event listeners must be attached to, Bing miner must be able to read parts of the DOM of sites hosted on `bing.com` so that it can search for the query form. Specifically, BingMiner must be able to obtain a handle on the element with id `sb_form` to attach an event listener, and subsequently on the element with id `sb_form_q` to read the search query.
3. BingMiner must be able to write data from `bing.com` to the personal store.

4.2.2 TwitterMiner

The functionality of TwitterMiner is similar to that of BingMiner; the user's interactions with Twitter are explicitly intercepted, analyzed, and used to update the user's interest profile. However, unlike BingMiner, TwitterMiner does not need to understand the structure of `twitter.com` pages or the user's interactions with them. Rather, it utilizes the RESTful API exposed by `twitter.com` to periodically check the user's twitter profile for updates. When the user posts a new tweet, TwitterMiner analyzes its content using REPRIV's classifier to determine how to update the personal store.

As shown in Figure 7, TwitterMiner needs only two capabilities from REPRIV, as the `twitter.com` API simplifies its task.

1. It must be able to make XHR-style requests to `twitter.com`. The second argument of the *CanCommunicateXHR* capability in Figure 7 indicates that TwitterMiner cannot send any sensitive information derived from the store, or a page visited by the user, in such a request.
2. It must be able to update the store to reflect data derived from `twitter.com`

The source code for TwitterMiner is shown in Figure 9, in both C# and Fine. One can see that the two versions are nearly identical in control flow and API usage. There are only two places in the Fine code in which the programmer must justify to the compiler that the stated policy is in fact being enforced. The first is in the type signature of `CollectLatestFeed`, where a refined type is used to tell the compiler that the identifier `extid` in fact refers to the extension ID stated in the policy manifest. The second location is the first statement in `CollectLatestFeed`, where a provenance label is constructed to reflect the source of information that will be collected by TwitterMiner, e.g. `twitter.com`. This allows the compiler to verify that the tracked information being sent to the store at the end of `CollectLatestFeed` is in accordance with the policy. Refinements on the type of API function `MakeXDocRequest` make it impossible for the programmer to forge this provenance label; if the constructed label does not accurately reflect the URL passed to `MakeXDocRequest`, a type error will indicate a policy violation.

4.2.3 NetflixMiner

NetflixMiner is a slightly more complicated example than the previous two. This extension performs two high-level tasks. First, it observers user behavior on `netflix.com`, and updates the personal store to reflect the user's interactions sites hosted on that domain. Second, it provides specific parties (`fandango.com`, `amazon.com`, and `metacritic.com`) with a list of the user's most recently viewed movies for a specific genre.

Figure 7 shows the capabilities needed by NetflixMiner to carry out these tasks.

1. It must be able to listen for `click` events on DOM elements with class labels `rv1` - `rv5`, as these indicate the rating the the user gives to a movie.

Extension	Policy
BingMiner	*CanHandleSites*(“bing.com”) *CanCaptureEvents*(“bing.com”, onsubmit) *CanReadDOMId*(“bing.com”, sb_form), *CanReadDOMId*(“bing.com”, sb_form_q) *CanUpdateStore*(*Tag*(“bing.com”, “bingminer”))
TwitterMiner	*CanCommunicateXHR*(“twitter.com”, ∅) *CanUpdateStore*(*Tag*(“twitter.com”, “twitterminer”))
NetflixMiner	*CanHandleSites*(“netflix.com”) *CanUpdateStore*(*Tag*(“netflix.com”, “netflixminer”)) *CanReadStore*(*Tag*(“netflix.com”, “netflixminer”)) *CanReadDOMClass*(“netflix.com”, rv1), *CanReadDOMClass*(“netflix.com”, rv2) *CanReadDOMClass*(“netflix.com”, rv3) *CanReadDOMClass*(“netflix.com”, rv4) *CanReadDOMClass*(“netflix.com”, rv5) *CanCaptureEvents*(“netflix.com”, onclick) *CanServeInformation*(“fandango.com”, *Tag*(“netflix.com”, “netflixminer”)) *CanServeInformation*(“amazon.com”, *Tag*(“netflix.com”, “netflixminer”)) *CanServeInformation*(“metacritic.com”, *Tag*(“netflix.com”, “netflixminer”)) *CanReadLocalFile*(“moviegenres.txt”).
GlueMiner	*CanCommunicateXHR*(“getglue.com”, (*Tag*(“netflix.com”, “netflixminer”)) *CanCommunicateXHR*(“getglue.com”, (*Tag*(“twitter.com”, “twitterminer”)) *CanCommunicateXHR*(“getglue.com”, (*Tag*(“facebook.com”, “facebookminer”)) *CanReadStore*(*Tag*(“twitter.com”, “twitterminer”)), *CanReadStore*(*Tag*(“netflix.com”, “netflixminer”)) *CanReadStore*(*Tag*(“facebook.com”, “facebookminer”)) *CanServeInformation*(“fandango.com”, *Tag*(“getglue.com”, “glueminer”)) *CanServeInformation*(“fandango.com”, *Tag*(“netflix.com”, “netflixminer”)) *CanServeInformation*(“linkedin.com”, *Tag*(“getglue.com”, “glueminer”)) *CanServeInformation*(“linkedin.com”, *Tag*(“twitter.com”, “twitterminer”)) *CanServeInformation*(“linkedin.com”, *Tag*(“facebook.com”, “facebookminer”))

Figure 7: Policies for sample REPRIV extensions.

Name	Lines of code		Verification
	C#	Fine	Time (s)
TwitterMiner	89	36	6.4
BingMiner	78	35	6.8
NetflixMiner	112	110	7.7
GlueMiner	213	101	9.5

Figure 8: Miner characteristics.

2. It must be able to update the personal store to reflect information derived from netflix.com pages, as well as read that information back at a later time.
3. It must be able to return information it can read from the personal store to requests from fandango.com, amazon.com, and metacritic.com.
4. It must be able to read from the local file “moviegenres.txt”, to associate movies in the personal store with genre labels given in requests from third-party sites.

Note that the policy is explicit about information flows: *only data computed by* NetflixMiner *can be communicated to a small number of third-party sites*. This degree of restrictiveness is necessary to ensure the privacy of the user’s sensitive information, without obliging the user to respond to modal access control checks at runtime.

4.2.4 GlueMiner

GlueMiner is different from previous examples in that it does not add anything to the store; rather, it provides a privacy-preserving conduit between third-party websites that want to provide personalized content, the user’s personal store information, and another third party (getglue.com) that uses personal information to provide intelligent content recommendations. The appendix shows GlueMiner’s source code. The function predictResultsByTopic is the core of its functionality, effectively multiplexing the user’s personal store to getglue.com: a third-party site can use this function to query getglue.com using data in the personal store. This communication is made explicit to the user in the policy expressed by the extension. Given the broad range of topics on which getglue.com is knowledgeable, it makes sense to open this functionality to pages from many domains. This creates novel policy issues: the user may not want information in the personal store collected from netflix.com to be queried on behalf of linkedin.com, but may still agree to allowing linkedin.com to use information from twitter.com or facebook.com. Likewise, the user may want sites such as amazon.com and fandango.com to use the extension to ask getglue.com for recommendations based on the data collected from netflix.com.

This usage scenario suggests a more complex policy for the proposed extension.

- The extension must only communicate personal store information from twitter.com and facebook.com to linkedin.com through the return value of predictResultsByTopic. Additionally, the information that is ultimately returned will be tagged with labels from getglue.com, as it was communicated to this host to obtain recommendations. Thus, GLUEMINER must be able to communicate these sources to getglue.com, and it must be able to send information tagged from getglue.com to linkedin.com through the return value of predictResultsByTopic.

```
using RePriv;

namespace TwitterMiner
{
 static class Program
 {
  static string userId;
  static List<string> guids;
  static RePriv.ExtensionPrincipal p;

  static void CollectLatestFeed(object source,
                                ElapsedEventArgs e)
  {
   // Get the user's twitter RSS feed
   TrackedValue<XDocument> twitFeed =
    RePriv.MakeXDocRequest("twitter.com",
     "http://twitter.com/.../"+userId+".rss", p);
   // Extract the latest tweet from the feed
   TrackedValue<string> cur = twitFeed.Bind(
    x => (from d in x.Descendants("item")
          where !guids.Contains(d.Element("guid"))
          select (string)d.Element("description")
          ).Take(1).Single());
   // Find the categories that apply
   TrackedValue<List<string>> cat =
    cur.Bind(computeQueryCategories);
   // Update the personal store
   RePriv.AddEntry(cur, "contents:tweet", cat);
  }

  static void Main()
  {
   guids = new List<string>();

   Timer feedTimer = new Timer();
   feedTimer.Elapsed +=
    new ElapsedEventHandler(CollectLatestFeed);
   feedTimer.Interval = 600000;
   feedTimer.Start();
  }
 }
}
```

```
module TwitterMiner

open Url
open RePrivPolicy
open RePrivAPI

// Policy assumptions
assume extid: ExtensionId "twitterminer"
assume PAx1: CanCommunicateXHR "twitter.com"
assume PAx2: forall (s:string) . (ExtensionId s) =>
  CanUpdateStore (P "twitter.com" s)

// Miner code
val GetDescription: xdoc -> string
let GetDescription d =
 let allMsgs =
 ReadXDocEls d "item" (fun x -> true) "description" in
 match allMsgs with
  | Cons h t -> h
  | Nil -> ""

val CollectLatestFeed: ({s:string | ExtensionId s}) ->
                       mut_capability ->
                       unit ->
                       unit
let CollectLatestFeed extid mcap u =
 let twitterProv = simple_prov "twitter.com" extid in
 let reqUrl =
        mkUrl "http" "twitter.com" "statuses..." in
 let twitFeed =
        MakeXDocRequest reqUrl extid twitterProv mcap in
 let currentMsg =
        bind twitterProv twitFeed GetDescription in
 let categories =
        bind twitterProv currentMsg ClassifyText in
 AddEntry twitterProv currentMsg "tweet" categories mcap

val main: mut_capability -> unit
let main mcap =
 let collect =
  (CollectLatestFeed "twitterminer" mcap) in
 SetTimeout 600000 collect
```

Figure 9: Twitter miner in C# (left) and Fine (right), abbreviated for presentation.

- Similarly, the extension must only leak information from `netflix.com` to `getglue.com` on behalf of `amazon.com` or `fandango.com`. This creates policy requirements analogous to those of the previous case.

The policy requirements of `GlueMiner` are made possible by REPRIV's support for multi-label provenance tracking. Note also the assumption that `getglue.com` is not a malicious party, and does not otherwise pose a threat to the privacy concerns of the user. This judgement is ultimately left to the user, as REPRIV makes explicit the requirement to communicate with this party, and guarantees that the leak cannot occur to any other party.

5. Experimental Evaluation

The experimental section is organized as follows. First, we characterize the performance overhead of REPRIV on browsing activities, with respect to both the default behavior mining that occurs in the background and the topic-specific extensions discussed in Section 4. Then, we talk about the quality of our document classifier, that is used for all default in-browser behavior mining. Finally, we discuss the usability concerns that arise with REPRIV.

5.1 Performance Overhead

We evaluated the effect of REPRIV on the performance of web browsing activities. Several aspects of REPRIV can affect the performance of browsing. This section is organized to provide a separate discussion of each such aspect: the effect of default in-browser behavior mining, the effect that each proposed personalization extension (Section 4) has on document loading latency, and the performance of primary extension functionality.

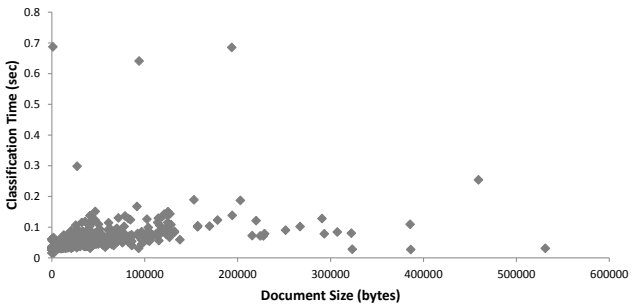


Figure 10: Document classification time

5.1.1 In-Browser Behavior Mining

One of the major components of REPRIV is the behavior mining that happens by default inside the browser, as the user navigates sites. In this section, we characterize the cost of performing this type of mining and the impact that it has on browser performance. Figure 10 depicts the amount of time in seconds needed by REPRIV to classify a document, plotted against the size of the document. Nearly all documents are classified in around one-tenth of a second; given this result, it is clear that REPRIV will not adversely affect the performance of the browser.

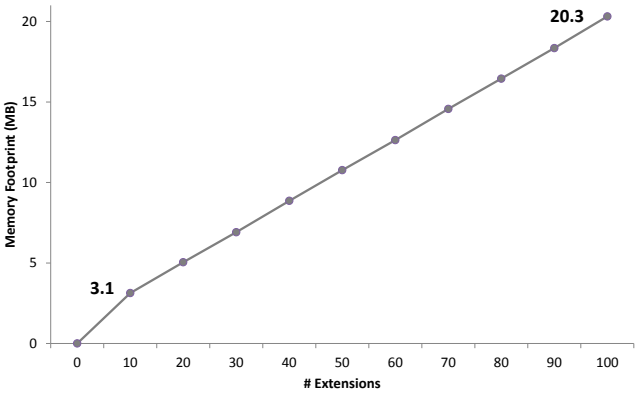


Figure 11: Miner memory footprint

5.1.2 Personalization Extensions

One concern with REPRIV's support for miners is the possibly arbitrary amount of memory overhead that it can introduce. We sought to characterize the memory requirements of REPRIV miners, by loading many compiled copies of the four miners presented in the previous section into a running instance of `C3`. Figure 11 shows the results. We see that in the extreme case of one hundred miners loaded into memory, only 20.3 megabytes of memory are needed.

5.2 Classifier Effectiveness

We sought to characterize the quality of the default in-browser classifier. However, doing so is not straightforward, as the task of document classification is inherently subjective. Our evaluation focuses on two metrics: the rate at which a user's interest profile converges, and human-perceived accuracy of the classifiers.

5.2.1 Profile Convergence

The rate at which a user's interest profile converges is an important property of our implementation, as it indicates the reliability of the personalization information provided by REPRIV. To measure the convergence of a profile, we require a notion of its final form. All of our measurements are taken over history traces of IE8 users, so the final profile that we use in these measurements is simply the profile computed by our classifier after processing an entire trace. All convergence measurements for a given trace are taken relative to the final profile for that trace, computed in this manner.

We use two measures of convergence. The first is the percentage of current entries in the top-ten list of interest categories that are also present in the final top-ten list. This measure is relevant because we forsee many websites querying REPRIV for top interests using the protocol outlined in Section 3. The second measure is the average distance of each interest category in the current ordering from its position in the final ordering; this gives a global view of interest profile stability.

The results of these experiments are presented in Figures 12 (a) and (b), which depict top-ten and distance convergence, respectively. They key point to notice about both of these curves is the state of the computed interest profile after 20% completion: 50% of the final top-ten categories are already present, and the global convergence curve has reached a point of gradual decline. This implies that the results returned by the core mining algorithm will not change dramatically from this point.

5.2.2 In-Browser vs. Public Data Mining

We claim that a major incentive for web service providers to utilize the personalization features enabled by REPRIV is the high quality of personal information that is available within the browser, relative to other types of information used for this purpose. In this subsection, we compare REPRIV's mining algorithm when used over browsing history data to the results obtained by gathering publicly-available information given a person's name. This approach is being used to facilitate personalization by a number of websites [31].

We see a fundamental problem with this approach, in that most names have several homonyms, and the precision and accuracy of a behavior profile will be adversely affected by this condition. To demonstrate this fact, we began by measuring the number of distinct homonyms for 48 names selected at random from a phone book. To take this measurement, we used a search engine called "WebMii" [34] which returns a listing of much of the publicly-available information about a particular name on the web, in addition to a list of homonyms for that name. The results are displayed in Figure 13 (a): each bucket on the x-axis contains all of the values between the listed number, and that immediately left of it. Noteworthy is the fact that fewer than ten of the names were found to be unique on WebMii; the remaining names either had no visible web presence, or from dozens to hundreds of homonyms. Clearly, these names would be very difficult to build an accurate profile for content personalization without additional input.

Figure 13(b) relates the confidence in result accuracy that REPRIV's core mining algorithm produces for documents collected by searching the web for documents with a given name, versus running the algorithm over a user's search history. The confidence is the sum of the probabilities computed for each interest category in the user's final top-10 interest profile, normalized by the number of documents used to build the profile to fit a scale of 0 to 1. The public profiles and user histories do not correspond to the same person when grouped at the same point on the x-axis; rather, they are sorted by confidence. To build a public profile for a given name, we searched for that name on `yahoo.com`, `facebook.com`, `twitter.com`, `hi5.com`, and `myspace.com`. The browsing histories are a subset of those used to compute the data in Figure 12. The results in figure 13(b) show that in all but a very few cases, the behavior mining algorithm was able to come to a much stronger conclusion given browsing histories.

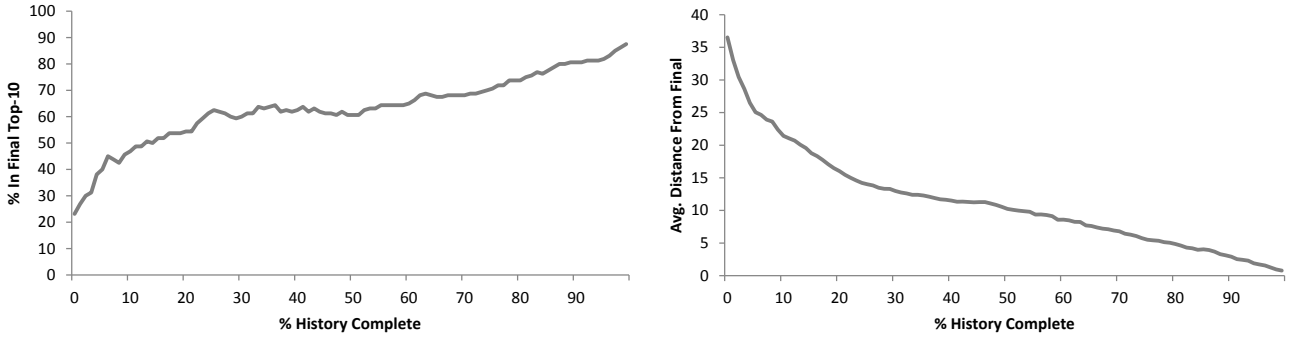
6. Case Studies

While the previous section provided a basic experimental evaluation of both the core mininig strategy and miners used in REPRIV, this section goes more in depth using two case studies, both evaluated on large quantities of real data. Section 6.1 talks about our search personalization experiment. Section 6.2 discusses news personalization.

6.1 Search Personalization

We wrote an extension that uses REPRIV's APIs to personalize the results produced by the main Bing search engine. The extension operates by observing the user's previous behavior on Bing, and memoizing certain aspects relevant to future searches. Specifically, for a given search term, the extension records which sites the user selected from the results pages, as well as the frequency with which each host is selected in search results (across all searches). When a new search query is submitted, the extension checks its history of previously-recorded searches for an identical match, and places the previously-selected results at the top of the current ranking. The remaining results are ranked by the frequency with which the user visits the host of each result.

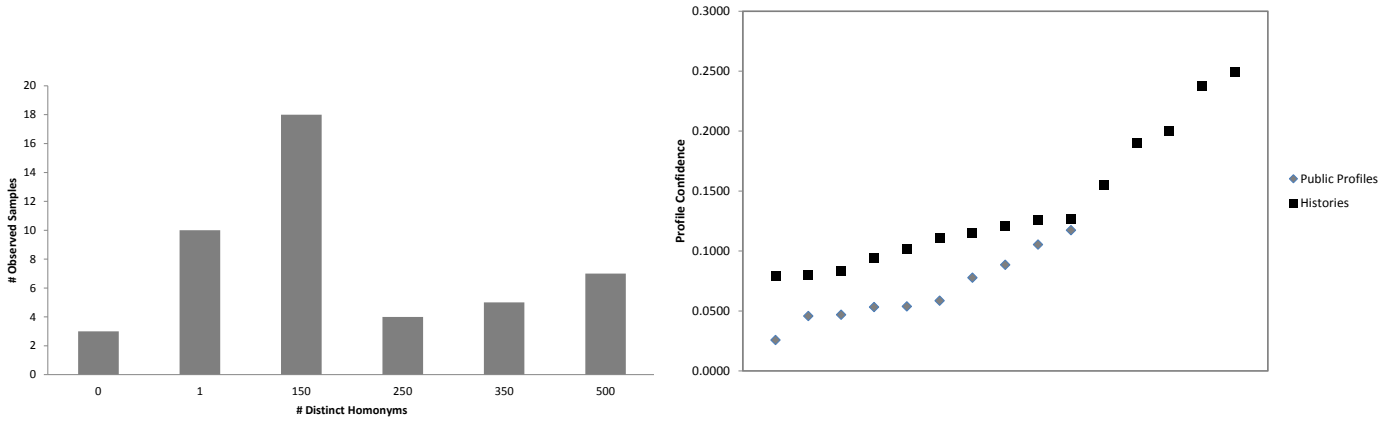
This type of search personalization is appealing for two reasons. First, the quality of results it provides is quite good, as discussed below. Second, it is not particularly invasive, as it requires observing user interaction on a single domain (`bing.com`). Furthermore, this information is leaked back to no site other than `bing.com` through re-arranging the result pages of queries submitted to the search en-



(a) Average top-ten category convergence curve.

(b) Average category distance convergence curve.

Figure 12: Convergence curves.



(a) Homonyms for 48 randomly-selected names

(b) Profile confidence

Figure 13: In-browser vs. public information-based personalization.

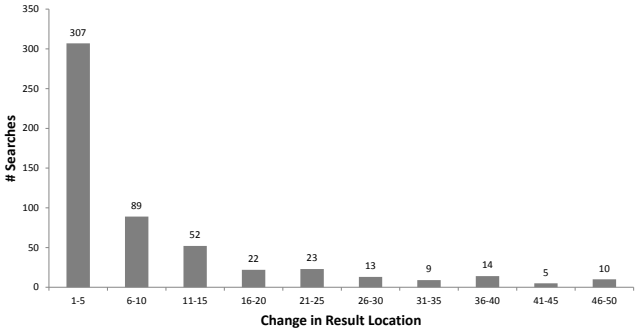


Figure 14: Search personalization effectiveness.

gine; if the user has cookies enabled, then `bing.com` learns this information by default. It is also important to note that information is only leaked to `bing.com` if the results pages contain JavaScript code that reflects on the layout of the DOM, and takes note of the relative position of search results. This activity would not be possible to hide from the Internet community, effectively minimizing its risk to end-user privacy and giving `bing.com` disincentive to do it.

To provide this functionality, the extension needs the following capabilities:

- To determine which search results the user selects from `bing.com` sessions, the extension must be able to receive `onclick` events from pages hosted by `bing.com`.
- To access a full list of search results over which it can perform re-ranking, the extension uses a public web API. For this, it must be able to make HTTP requests to either `bing.com`, `yahoo.com`, or `google.com` (search API providers).
- To re-arrange the results pages from `bing.com`, the extension must be able to change the `TextContent` of HTML elements on `bing.com`, as well as well as call change the `href` attribute of `a` elements.
- To memoize search engine interactions, the extension must be able to write data from `bing.com` to the personal store.

6.1.1 Implementation Details

We implemented the extension for `C3` as 382 lines of Fine.

The code is presented in Section B.2. The extension uses several of the API's exposed by REPRIV: `XMLHttpRequest`, `SetAttribute`, `SetTextContent`, `GetElementById`, and `GetChildren`.

When loaded into the browser, the extension requires approximately 200KB of memory.

6.1.2 Experimental Methodology

To evaluate the effectiveness of search personalization, we utilized the histories of nineteen users of the Bing search toolbar. Each history represents seven months of Bing search activity. Our methodology for evaluating the effectiveness of search personalization algorithm is based on the results selected by users for a given query. For each search performed by a particular user, we split the search history into two chronologically-contiguous halves. We construct the relevant portions of a personal store needed to perform search personalization using the first half, and use the second half to evaluate the effectiveness of the algorithm. For each query in the second

half of each trace, we evaluated the effectiveness of our search personalization algorithm as follows:

1. Submit the query to the Yahoo BOSS API [36], and collect the default search result ranking.
2. Re-rank the results according to the algorithm discussed above.
3. Note the difference in position for the search result selected by the user between the default and personalized rankings. A positive difference indicates that the selected result is ranked higher in the personalized results, whereas a negative difference indicates the opposite.

This process simulates the user's interaction with a personalized and non-personalized search engine, giving us a baseline for comparison.

6.1.3 Evaluation

The results of our evaluation are summarized in Figure 14. This histogram shows the number of positions the user's selected result moved towards the top of the ranking when the search personalization extension was able to improve results.

We found that for a given user, *the extension was able to improve results 49.1% of the time by raising the user's selected result 8.2 positions toward the top, on average*. 7.7% of the time, the extension lowered the ranking of the user's selected result, but when this occurred, the result was moved downwards an average of only 2.4 positions. For the remaining percentage of time, 43.2%, the extension had no effect on the ranking of the user's selected result. These results show that our search personalization algorithm is able to provide useful functionality for a large portion of the user's web searching activities, while giving the user explicit control over the way in which personal information is used in the process.

6.2 News Personalization

We wrote an extension that uses REPRIV's computed behavior profile to personalize the New York Times front page. The extension utilizes the collaborative filtering provided by the `digg.com` community by matching the user's top interest categories with topic names understood by `digg.com`, and periodically querying its web API for "hot" stories in those topics. When the user visits `nytimes.com`, New York Times articles cached from `digg.com` API queries are presented at the top of the page, in place of the default headlines.

To perform this personalization, the extension needs several capabilities.

- To query the `digg.com` API, it must be able to send HTTP requests to `digg.com` and access the formatted responses containing news stories.
- To locate the appropriate HTML elements on the `nytimes.com` front page for personalized re-formatting, the extension must be able to call `GetElementById` and `GetAttribute("class")` on DOM nodes hosted by `nytimes.com`.
- To re-format the `nytimes.com` front page, the extension must be able to change the `TextContent` of nodes on `nytimes.com` nodes, as well as call `SetAttribute("href")` on them.
- To construct te appropriate query to `digg.com`, it must be able to query the personal store to learn the top interests of the user.

6.2.1 Implementation Details

We implemented the extension for `C3` as 124 lines of Fine. The code is presented in Section B.1.

The extension uses several of the API's exposed by REPRIV: `XMLHttpRequest`, `GetAttribute`, `SetAttribute`, `SetTextContent`, `GetTopInterests`, `GetElementById`, `SetTimeout`, and `GetChildren`.

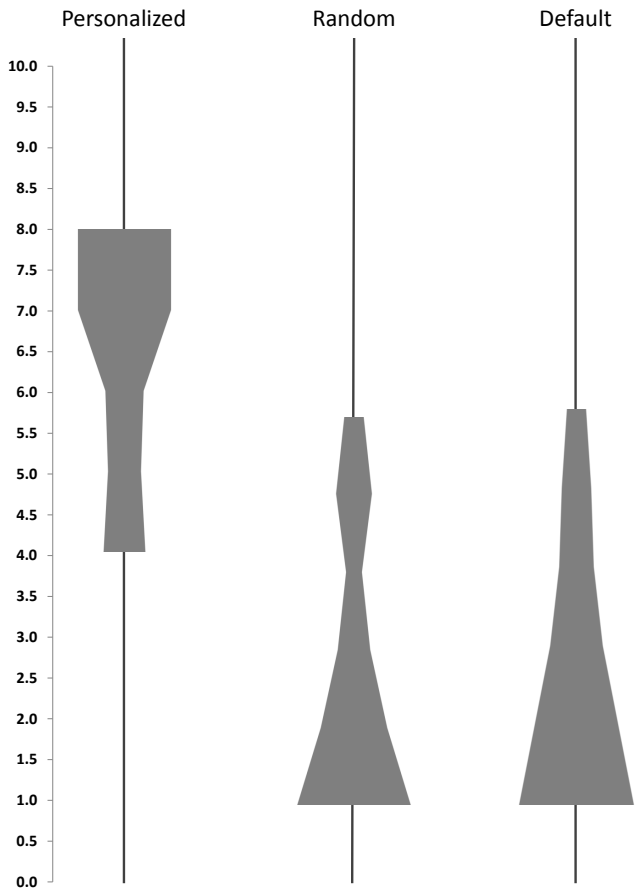


Figure 15: News personalization effectiveness.

When loaded into the browser, the extension requires approximately 200KB of memory.

When navigating to `nytimes.com`, we found that the extension introduced a latency of 6% over the default loading time without any personalization, which is a consequence of the fact that the extension modifies the DOM after initial loading is complete. This overhead does not reflect the time needed to query the `digg.com` API, which occurs periodically in a background thread that runs when the CPU is otherwise idle.

6.2.2 Experimental Methodology

We performed a set of experiments using Amazon's Mechanical Turk service [23] to demonstrate that our news personalization system does not trivialize the problem of delivering personalized content in fulfilling the goal of preserving user privacy. In other words, we sought to show that the type of personalization offered by our extension is relevant to internet users.

To do so, we generated 1,920 artificial behavior profiles. 900 of the profiles contained three randomly-selected user interest topics, and the rest contained three topics related by the same top-level ODP category. This distribution models users with both focused and diverse interests. We then seeded our personalization algorithm with each profile, and captured an image of the stories that would be presented by the extension. The image contained the headline of each story, as well as a short summary of each story, in a manner similar to the default `nytimes.com` layout.

Using the images and interest profiles, we generated a set of Mechanical Turk surveys. Each survey consisted of twelve questions, where each question paired a news content image with a po-

tential behavior profile, and asked the user how relevant the stories presented in the image were to the given set of interest topics, on a scale of 1 to 10. For each survey, approximately half of the questions matched the image with the interest profile our algorithm used to generate them, and the other half were paired randomly. Each survey contained an additional question that paired the default `nytimes.com` front page stories with a random interest profile. The latter two pairings served as our control, to determine how relevant users found hypothetical interest profiles to general news stories.

6.2.3 Evaluation

Figure 15 is a violin plot that shows the results of our Mechanical Turk experiments. Each column of the plot summarizes the distribution of responses for a particular pairing of news articles and behavior profiles.

"Personalized" denotes real pairings of personalized news stories to behavior profiles, "Random" refers to pairings of news stories to randomly-generated behavior profiles that do not bear a meaningful connection, and "Default" denotes the stories presented on the default `nytimes.com` front page paired with a random behavior profile. For each column, the statistical mean among survey responses, as well as the surrounding vicinity of one standard-deviation, is plotted.

As the figure indicates, respondents gave stories personalized with our algorithm significantly higher relevance scores than the control samples. For personalized content, ratings between 6.5 and 8 recieved the most responses, with markedly lower variance than the control. While some overlap in response exists between personalized content and the control, the majority of control responses mass around low relevance scores, indicatating a clear improvement in percieved relevance for content personalized using our algorithm.

In summary, the results of this news personalization experiment show that REPRIV enables useful and effective personalization of news content without sacrificing control over private information.

7. Discussion

So far, we have showed how REPRIV presents an alternative to intrusive end-user tracking to support personalization on the web, by providing a mechanism on the host that users can opt to use. We have also discussed some of the technical aspects of REPRIV that allow it to provide the user with explicit and precise control over the release of this information to third parties. In this section, we begin with a discussion of the incentives that users, service providers, and developers have to adopt REPRIV (Section 7.1). We then discuss some concerns that arise regarding the usability of REPRIV, and a distribution model that eases some of these concerns (Section 7.2). We then present a brief discussion of collaborative mining and filtering algorithms in REPRIV, which are becoming more popular in web applications (Section 7.3). Finally, we discuss the implications of REPRIV for the privacy modes that are now present in nearly all popular browsers (Section 7.4).

7.1 Incentives

It may not be clear that service providers and developers have sufficient cause to utilize and contribute to REPRIV. Here we discuss why this is not the case, and why REPRIV naturally complements forces currently present on the web.

7.1.1 Users

The incentives for users to adopt REPRIV are immediate: REPRIV was designed to facilitate the types of personalized web experience that have become popular today, while allowing users to maintain control of their personal information. Beyond better privacy, this has additional implications for improved user experience. First, because all of the personal information needed to enable personalization is maintained on the client, the user can share it with any party that can provide a better experience for the user. This means that the so-called *cold-start problem*, where a user visits a new website and is not able to recieve personalized content for lack of data, is no longer an issue. Secondly, users who prefer more personalization are free to install arbitrarily sophisticated miners that allow service providers to create more advanced content. Effectively, REPRIV allows users to opt for the level of sharing and personalization that they find most appropriate. Finally, we have demonstrated that REPRIV's performance overhead is minimal, so there is very little disincentive for a user to adopt REPRIV.

7.1.2 Service Providers

An important consideration is the incentive that service providers have to utilize the personal information maintained by REPRIV, rather than continuing to collect data invasively via mechanisms such as third-party tracking. It may seem unreasonable to expect service providers to voluntarily switch to a system like REPRIV at this point, having built substantial momentum with existing tracking techniques. While a truly anonymous browsing mode would leave content providers without an alternative, we assert that incentives already exist for service providers to adopt REPRIV without the need for such measures. The first such incentive is the quality of information that REPRIV can provide relative to other techniques. REPRIV gives service providers the opportunity to utilize data that is not impeded by third-party cookie blockers or cookie deletion on the client, that is derived using information from the user's complete browsing experience. Secondly, because REPRIV gives content providers a way to personalize that respects user privacy, content providers have the opportunity to differentiate themselves from competitors on the basis of respecting end-user privacy. The fact that users are more likely to select services that do not seem invasive creates incentive for content providers to abandon existing data collection measures.

7.1.3 Miner Developers

Another matter to consider is the incentive to write miners for REPRIV. We forsee a number of likely scenarios. First, online businesses have direct incentive to write miners that allow them to collect better information about the user. For example, the operators of `blockbuster.com` have clear incentive to write a miner that collects data about user behavior on `netflix.com`, `fandango.com`, *etc.*, and returns it to `blockbuster.com`. However, REPRIV does not provide a mechanism for a particular content provider to disallow miners from collecting behaviors pertinent to that provider, aside from an appeal to the miner distribution framework discussed in Section 7.2. Rather, REPRIV leaves it to the user to decide whether to allow a miner to observe his interaction with a particular website. This may create tension between competing businesses, that do not consider this sort of behavior to be in their best interest; it is the philosophy of REPRIV that this choice ultimately falls to the user.

Another likely scenario arises with general content recommendation web services and social networks, such as `getglue.com` and `hunch.com`. These sites allow users to create detailed profiles of their likes and interests for the purpose of sharing them with other users and recieving content recommendations. Key to the effectiveness of these services, and thus the ultimate success of the website, is that a large amount of personal information about users is present to use as the basis of recommendation. REPRIV miners are a safe and effictive way for these sites to realize this goal.

Finally, it is important not to overlook the contributions made by enthusiasts to the extension libraries of Firefox, Apple, Win-

dows Live, and similar services. While some of the extensions in these libraries are intended to provide revenue for companies, a surprisingly large number of them are not. Rather, they are written and maintained by either open-source developers or enthusiasts for various reasons, and are in many cases competitive with those associated with businesses. We believe that this phenomenon has potential in the context of REPRIV.

7.2 Usability Concerns & Distribution Model

A primary goal of REPRIV is to be as explicit as possible about the transfer of sensitive user data. This means that at some point, the user must manually consent to the information being disseminated by REPRIV to remote parties. For core mining data, this is not particularly challenging. In fact, the structure of the information produced by core mining was designed to be highly informative to content providers and intuitive for end-users: when prompted with a list of topics that will be communicated to a remote party, most users will understand the nature and degree of information sharing that will subsequently take place if they consent.

There is a rich body of research that considers the question of how best to present such prompts to the user [20], and one area that we would like to explore in future work is incorporating these findings into REPRIV. There is an additional danger of overwhelming the user with prompts for access control, effectively de-sensitizing the user to the problems addressed by the prompts. One way to reduce the interactive burden is to allow the user to tell REPRIV to remember their response for a particular domain. However, this is not likely to suffice as a final solution, so we would additionally like to explore more expressive policy frameworks for specifying these preferences as part of our ongoing work.

On the other hand, the usability problems posed by miners is more difficult. While the privacy policies imposed on miners are expressive and precise, it is difficult to make their implications explicit to an average user. Simply put, it is unreasonable to expect a user to understand a first-order logical formula stated in terms of host names and provenance labels.

Because there is no technical mechanism that can test a miner for *all* undesirable characteristics (e.g. performance, usability, etc.) up front, we suggest a distribution model that allows for miners to be revoked from circulation. This model is similar to that adopted by Firefox, Apple, and Symbian for supporting third-party functionality. In such a model, miners are submitted to a central repository for review before they are made available for installation. The owner of such a repository is expected to posess considerably more technical sophistication than most browser users. When the repository owner reviews a miner, he can first verify that it follows its stated policy. The properties of Fine make this step both fast and trivial for the reviewer, whereas this step can take weeks or even months in the case of Firefox's extension library of Apple's app store. Additionally, the reviewer can perform a sanity check to verify that the miner's policy, as well as the complexity of its code, corresponds to its stated functionality. Finally, if at some point in the lifetime of a miner it becomes clear that it is abusing access to sensitive user data, the reviewer can remove the miner from the library and notify all users who previously installed it.

7.3 Collaborative Mining and Filtering

Not currently addressed in REPRIV are the needs of explicit collaborative mining algorithms [6], [18] that factor in the preferences of a community as a basis for conclusions about a particular user. Currently, REPRIV can be used to provide this functionality through a miner that sends all of the relevant information, which is likely to be a large volume, to a central server for collaborative mining. However, this is not as attractive from a privacy standpoint as other collaborative mining schemes designed with privacy in mind [22] [38].

The problem with implementing these schemes using REPRIV is that their privacy-preserving characteristics cannot be mechanically verified using REPRIV policies; the same REPRIV policy would apply to a privacy-preserving data mining algorithm as a miner that simply sends the needed information to a third party. In the future, we will explore methods for facilitating this type of functionality either at the API level or through additional verification of miner code. A number of recent advances, such as differential privacy [7] and quantitative information flow [19], suggest promising directions for this problem.

7.4 Anonimization, Blocking Techniques and Privacy Modes

Recently, major browsers have come to support some form of a "private browsing mode" [2]. Although the precise meaning of this term varies between browsers, the basic idea behind this feature is to prevent websites from reading persistent data such as cookies for a particular session, while also erasing the persistent data for that session when the user terminates it. There have been a number of other browser add-ons and modifications that attempt to anonymize the user on the web; an incomplete list includes TrackMeNot [13], Torbutton, SafeCache [29], SafeHistory [29], and IE8 InPrivate browsing. While it is clear that a truly anonymous browsing mode would force content providers to use REPRIV, no such mode has been successfully implemented [2], and it is not clear that doing so is technically feasible [8]. However, we assert that REPRIV does in fact facilitate end-user privacy on the web, by creating incentives for content providers to use privacy-sensitive personalization techniques, rather than relying on the invasive collection mechanisms currently available. In this respect, REPRIV is complementary to private browsing modes; it provides a mechanism for allowing personalized content without the need for the tracking mechanisms currently used by content providers, which are not compatible with anonymous browsing.

Because REPRIV is currently implemented within an experimental browser that does not support private browsing, it does not contain a mechanism for adhering to such a constraint. However, the basic architecture of REPRIV is amenable to privacy mode, as the only persistent state it maintains resides within a single database table. When private browsing mode is entered, REPRIV's database is transitioned to a copy-on-write state; when the session ends, the copy is erased, if it exists.

8. Related Work

Related work spans privacy issues in the following broad categories: targeted advertising, web applications, private browser state, private browsing modes, and web personalization, which we cover in Sections 8.1–8.5, respectively.

8.1 Privacy in Advertising

The high-level problem of managing web users' desire for privacy and the associated trade-offs in personalized content delivery has been explored by others in various forms. One problem that has received much recent attention is that of delivering targeted advertisements to web users without unduly violating their privacy. Freudiger *et al.* [9] observe that the prevalent mechanism for targeting advertisements to individual users is the *third-party cookie*. Third-party cookies allow advertisers to track user behavior across multiple sites in different domains without their explicit consent, and thus pose an immediate threat to privacy. Freudiger *et al.* propose a browser extension that allows users to directly manage third-party cookies and their visibility with respect to individual sites. Through proper management of cookies, users are free to decide the degree to which advertisers are able to track them, effectively giving them more control of their personal data. However, unlike with REPRIV,

this solution does not give users arbitrary, fine-grained control over the type of information that is given to third-parties. Furthermore, advertisers have no incentive to obey the privacy safeguards instantiated by this mechanism.

Guha *et al.* [12] present Privad, an architectural solution to the problem of privacy-preserving targeted advertisements. The key to Privad's privacy guarantees derives from the fact that all information needed to target and display advertisements is collected and stored on the user's local machine. Clearly, there are functional difficulties that arise because of this design choice. The first such difficulty is click-fraud: the advertiser has no immediate protection, as the display is anonymized for privacy. To remedy this concern, Privad relies on a semi-trusted *dealer* to anonymize user click behavior, while accurately reporting such activity to the advertiser for accounting purposes. The second issue addressed by Privad is scaling the privacy-preserving mechanism to realistic levels; this is currently a topic of ongoing research. There are a few key differences between REPRIV and Privad. Perhaps the most important difference is that REPRIV does not necessitate large-scale architectural shifts to accommodate strong privacy guarantees. Instead, REPRIV provides a simple client-side mechanism for controlled, user-directed dissemination of selected types of private data as needed by applications.

Toubiana *et al.* [33] discussed Adnostic, a solution similar in many ways to Privad. In Adnostic, user behavior is monitored locally on the client in order to build a model suitable for selecting targeted advertisements. Whenver an advertisement is to be displayed, Adnostic contacts the ad network to obtain a fixed list of potential advertisements. Based on the computed behavior profile, software on the client selects one ad for display, thus leaving all remote parties ignorant of the choice. Homomorphic encryption is used to charge Advertisers for individual renderings in a privacy-friendly way. However, Adnostic does not consider that a breach of user privacy has occurred if the advertiser learns when a user clicks on a particular advertisement, so click-based accounting can proceed as it currently does without privacy considerations. It is because of this assumption that Adnostic sidesteps the need for a semi-trusted dealer to enable proper accounting of advertisements between the user and advertiser. Like Adnostic, REPRIV observes user behavior on the client to build a model of user interests. However, the similarities between Adnostic and REPRIV end there, as the manner in which private information is disseminated in REPRIV allows it to be put to a wider range of uses than Adnostic, which is suitable only for advertisements.

Juels [17] was among the first to tackle the problem of preserving privacy with targeted advertising. In addition to proposing several simple schemes that allow some forms of personalized targeting, he described a scheme that utilizes private information retrieval (PIR) and mix networks to obtain strong privacy guarantees while allowing flexibility among targeting mechanisms. Essentially, advertisers provide a "'negotiation function"' that assigns advertisements to user profile types. Because the scheme relies heavily on strong cryptographic primitives, it does not scale to real-time demands, so Juels recommends the use of prefetching and caching on the client to meet users' performance expectations.

Provost *et al.* [28] observe that users' social graphs can provide insight into their interests that may be useful for advertisers. They proposed a somewhat privacy-preserving technique for constructing social graphs, based on identifying users who visit the same user-generated content sites. However, because they use third-party cookies to collect this data without first notifying users or obtaining their permission, some might dispute the degree to which their system protects privacy.

8.2 Privacy for Web Applications

As a reaction to the perceived decrease in privacy on the web, many have started exploring techniques that can be applied to restore some degree of privacy while still allowing for the rich web applications that people have come to expect. Jakobsson *et al.* [15] considered the problem of third-party sites mining users' navigation history. They posit that while most users are not comfortable sharing their entire history with content providers, they may be comfortable releasing *aggregate* information that lets a third party learn whether they have visited at least one or all sites from a large list of candidate sites. To that end, they developed a system that allows third parties to learn this type of information about users' navigation histories, but nothing more. All privacy assurances offered by this system derive from the fact that its mechanism is easily auditable by end-users, so parties who wish to mine history data have disincentive to cheat.

Becker and Chen [4] studied the feasability of inferring specific attributes of individuals based on their friend connections on social networking sites. They found that it is possible to deduce users' personal characteristics with high probabily, for many types of personal information. Worse yet, they found that it is very difficult to defend against this type of inference, assuming an attacker has access to the user's social graph. On average, they found that users would have to remove on the order of hundreds of friends from their connections in order to ensure the privacy of their own characteristics.

Narayanan and Shmatikov [25] studied the privacy implications of social networking for end-users. Their observation is that the operators of online social networking sites are sharing user data with commercially-interested third parties at an increasing rate, and are doing so after scrubbing the data of personally-identifying information in an alarmingly ad-hoc fashion. Relating users' privacy in a social network to node anonymity in the social network graph, they developed a *re-identification* algorithm that attempts to identify particular users in an anonymized social network. They found that if a user subscribed to both Twitter and Flickr, then the algorithm can correctly identify them with 88% accuracy.

McSherry and Mironov [22] attempted to restore a certain degree of privacy to collaborative recommendation algorithms, such as that used by Netflix. Citing the work of Narayanan and Shmatikov [24] in de-anonymizing users who take part in such systems, they worked in the framework on *differential privacy* [7] to build a collaborative recommendation algorithm that preserves the privacy of each individual rating entered by a participating user. Their algorithm was able to provide recommendations of quality comparable to that of the original Netflix recommendation algorithm.

8.3 Managing Private Browser State

A number of researchers have studied the manner in which modern browsers maintain private state, and how it relates to user privacy. McKinley [21] examined the privacy modes of popular browsers, as well as their ability to dutifully clear private state when explicitly directed by the user. She found that while some browsers do in fact clear private state when instructed, none of the browsers' privacy modes performs as advertised; each browser left some form of persistent state that could be later retrieved by web pages in different browsing sessions. In particular, third-party browser extensions posed a significant challenge, as they have the ability to write to disk without mediation by the browser.

A number of researchers have taken notice of the fact that the W3 standard mandates functionality, through CSS and JavaScript, that allows an untrusted web content provider to learn the presence of specific entries in the user's navigation history [16]. Janc and Olejnik [16] performed an in-depth study of this issue, and came to

the disturbing conclusion that 76% of internet users are vulnerable to the problem, while attackers can query up to 30,000 history entries per second on modern hardware configurations. Wondracek *et al.* [35] exploited this capability to completely de-anonymize users of popular social networking sites. Jackson *et al.* [14] observed this problem, and posit that the underlying problem is that browsers do not extend the same-origin policy to the navigation history state leveraged in the attack. They presented a Firefox extension that extends the same-origin policy in the required way, thus neutralizing the privacy threat; their solution applies to data in the browser cache as well, as they point out that this information can also be used to learn about the user's history. Recently, Mozilla has taken steps to prevent history sniffing [32], at the cost of sacrificing certain style features of web applications.

Taking a broader view of the problem, Eckersley [8] found that the problem of managing personally-identifying information in the browser is significantly more difficult than previously thought. The issue he finds is with a technique dubbed *browser fingerprinting*, wherein a large number of publicly-visible but otherwise benign browser attributes are combined to produce a nearly-unique identifying string for a particular browser. Among the attributes included in his proposed fingerprint are the user agent string, the HTTP accept headers, screen resolution, and list of installed plugins. He found that across all browsers which visited his site at `eff.org`, only one in 286,777 browsers will share a fingerprint with any single browser. Even worse, among browsers with both Flash and Java enabled, 94.2% had unique fingerprints.

8.4 Privacy-Preserving Browsing

Noting the tendencies of various third-parties to track users' browsing behavior without explicit permission, several researchers have approached the technical problem of maintaining user anonymity while browsing. Howe and Nissenbaum [13] created TrackMeNot, a Firefox extension that attempts to anonymize search behavior by periodically submitting random search queries to major search engines. Additionally, TrackMeNot supports a few mechanisms to simulate further aspects of normal user interaction with search engines, such as click-through and "'burst-mode"' queries, to decrease the likelihood that search engines will be able to reliably differentiate between artificial requests submitted on behalf of TrackMeNot, and real requests submitted by the user.

8.5 Web Personalization and Mining

Many have considered the possibility of automatically personalizing web content for users based on their interests, preferences, and behavior. The basis on which personalization is performed varies from application to application. Pierrakos *et al.* [27] surveyed the topic of mining users' behavior on a set of web services to infer information that will aid personalization. They found that almost all web personalization efforts fall into one of four broad categories: *(1)* memorizing information about the user to be later replayed, *(2)* guiding the user towards information in which they are more likely to be interested, *(3)* customizing layout or content to better match users' interests, and *(4)* supporting users' efforts to complete certain tasks. REPRIV is designed primarily to support the implementation of *(2)* and *(3)*, but it is not difficult to see how it can be used to support certain aspects of all types of personalization.

There are several browser add-ons popularly referred to as *toolbars* that perform data collection and user behavior mining. Perhaps the longest-running and most popular among them is the Alexa Toolbar [3], which for each user collects a complete browsing history, search engine query list, and summary of the advertisements presented to the user. This information is transmitted back to Alexa, where it is used to compute a number of analytic functions, some of which are returned to toolbar users as a service. Among the analytics are traffic statistics (including a comprehensive, internet-wide ranking of popular sites), related search queries for particular URL's, audience demographics, related links, and clickstream statistics. Until recently, Alexa made much of this information available to the public via a web API. Similarly, Bing [5], Google [11], and Yahoo [37] all offer toolbars, although they vary in the amount of mining and automatic personalization that they perform. The Yahoo and Google toolbars collect search engine and navigation histories, and consult them on new searches to present personalized results.

9. Conclusions

This paper presents REPRIV, an in-browser approach that aims to perform personalization without sacrificing user privacy. REPRIV accomplishes this goal by requiring explicit user consent in any transfer of sensitive user information. We showed how efficient and effective behavior mining can be added to a web browser to automatically infer the information needed to facilitate many personalized web applications, and evaluated this mechanism on real-world data. We also showed how third-party code can be incorporated into the system, and given access to sensitive user information, without sacrificing control and the possibility of user consent. Finally, we presented two end-to-end case studies of useful personalized applications, that showcase the abilities of REPRIV. We evaluated several aspects of these case studies over data collected from real browsing sessions, as well as human participants. Given our results, we are able to conclude that REPRIV allows a wide range of personalized web applications to exist, without requiring the user to sacrifice control over their personal information: personalized content and privacy can coexist on the web.

Acknowledgments

We wanted to thank several researchers for their help with various aspects of this project. This list is in no particular order: Sue Dumais and Jaime Teevan for helpful conversations on personalization, Qiang Wu for his help with document classification, Peter Bailey for providing insight into search personalization, Nikhil Swamy, Arjun Guha, and Jean Yang for their insights and experience with Fine, AJ Cannon and Michael Elizarov for sharing their opinions on the advantages and pitfalls of personalization, and David Molnar for his input and assistance throughout the course of this work.

References

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[2] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proc. of 19th Usenix Security Symposium*, 2010.

[3] The Alexa Toolbar. `http://alexa.com/toolbar`.

[4] J. Becker and H. Chen. Measuring privacy risk in online social networks. In *Proceedings of the 2009 Workshop on Web 2.0 Security and Privacy*, 2009.

[5] The Bing Toolbar. `http://www.discoverbing.com/toolbar`.

[6] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007.

[7] C. Dwork. Differential privacy: a survey of results. In *TAMC'08: Proceedings of the 5th international conference on Theory and applications of models of computation*, 2008.

[8] P. Eckersley. How Unique Is Your Web Browser? Technical report, Electronic Frontier Foundation, 2009.

[9] J. Freudiger, N. Vratonjic, and J.-P. Hubaux. Towards Privacy-Friendly Online Advertising. In *IEEE Web 2.0 Security and Privacy (W2SP)*, 2009.

[10] Google AdSense privacy information. `http://www.google.com/privacy_ads.html#toc-faq`.

[11] The Google Toolbar. `http://toolbar.google.com`.

[12] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving Ads from localhost for Performance, Privacy, and Profit. In *Proceedings of Hot Topics in Networking (HotNets)*, New York, NY, 2009.

[13] D. C. Howe and H. Nissenbaum. TrackMeNot: Resisting surveillance in web search. In I. Kerr, V. Steeves, and C. Lucock, editors, *Lessons from the Identity Trail: Anonymity, Privacy, and Identity in a Networked Society*, chapter 23, pages 417–436. Oxford University Press, 2009.

[14] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, 2006.

[15] M. Jakobsson, A. Juels, and J. Ratkiewicz. Privacy-Preserving History Mining for Web Browsers. In *Proceedings of the 2010 Workshop on Web 2.0 Security and Privacy*, 2010.

[16] A. Janc and L. Olejnik. Feasability and real-world implications of web browser history detection. In *Proceedings of the 2010 Workshop on Web 2.0 Security and Privacy*, 2010.

[17] A. Juels. Targeted advertising ... and privacy too. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 408–424, London, UK, 2001. Springer-Verlag.

[18] Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Trans. Knowl. Discov. Data*, 2010.

[19] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.

[20] A. M. Mcdonald, R. W. Reeder, P. G. Kelley, and L. F. Cranor. A comparative study of online privacy policies and formats. In *PETS '09: Proceedings of the 9th International Symposium on Privacy Enhancing Technologies*, 2009.

[21] K. McKinley. Cleaning Up After Cookies Version 1.0. Technical report, ISEC Partners, 2010.

[22] F. McSherry and I. Mironov. Differentially private recommender systems: building privacy into the net. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009.

[23] Amazon Mechanical Turk. `https://www.mturk.com/mturk/welcome`.

[24] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[25] A. Narayanan and V. Shmatikov. De-anonymizing social networks. *IEEE Sympolsium on Security and Privacy*, 2009.

[26] The Open Directory Project. `http://dmoz.org`.

[27] D. Pierrakos, G. Paliouras, C. Papatheodorou, and C. D. Spyropoulos. Web usage mining as a tool for personalization: A survey. *User Modeling and User-Adapted Interaction*, 13(4), 2003.

[28] F. Provost, B. Dalessandro, R. Hook, X. Zhang, and A. Murray. Audience selection for on-line brand advertising: privacy-friendly social network targeting. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009.

[29] Same origin policy: Protecting browser state from web privacy attacks. `http://crypto.stanford.edu/safecache/`.

[30] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *In Proceedings of the European Symposium on Programming (ESOP)*, pages 529–549, 2010.

[31] TargetAPI. `http://www.targetapi.com`.

[32] The Mozilla Team. Plugging the CSS History Leak. `http://blog.mozilla.com/security/2010/03/31/plugging-the-css-history-leak`, 2010.

[33] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2010.

[34] WebMii: A person search engine. `http://www.webmii.com`.

[35] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *31st IEEE Symposium on Security and Privacy*, 2010.

[36] Yahoo! BOSS API. `http://developer.yahoo.com/search/boss/`.

[37] The Yahoo Toolbar. `http://toolbar.yahoo.com`.

[38] J. Zhan, L. Chang, and S. Matwin. Privacy-preserving collaborative data mining. In T. Young Lin, S. Ohsuga, C.-J. Liau, and X. Hu, editors, *Foundations and Novel Approaches in Data Mining*, Studies in Computational Intelligence. Springer Berlin / Heidelberg, 2006.

A. REPRIV Miners

This section lists the following REPRIV miners:

- Netflix miner in Section A.1
- Twitter miner in Section A.2
- Bing miner in Section A.3
- Glue miner in Section A.4

as well as the case studies (Sections B.1 and B.2) and a large portion of the REPRIV API (Section C).

A.1 Netflix Miner

```
// NetflixMiner - Fine version

module NextflixMiner

open RePrivPolicy
open RePrivAPI
open Url

// Policy assumptions
assume PAx0:        ExtensionId  "netflixminer"
assume PAx1:        forall (s:string) . (ExtensionId s) => CanUpdateStore (P "netflix.com" s)
assume PAx3:        forall (s:string) . CanReadDOMId "netflix.com" s
assume PAx6:        CanReadDOMClass "netflix.com" "rv1"
assume PAx7:        CanReadDOMClass "netflix.com" "rv2"
assume PAx8:        CanReadDOMClass "netflix.com" "rv3"
assume PAx9:        CanReadDOMClass "netflix.com" "rv4"
assume PAx10:       CanReadDOMClass "netflix.com" "rv5"
assume PAx11:       CanCaptureEvents "onclick" (P "netflix.com" "netflixminer")
assume PAx12:       CanServeInformation "fandango.com" (P "netflix.com" "netflixminer")
assume PAx13:       CanServeInformation "amazon.com" (P "netflix.com" "netflixminer")
assume PAx14:       CanServeInformation "metacritic.com" (P "netflix.com" "netflixminer")
assume PAx15:       CanHandleSites "netflix.com"
assume PAx16:       CanReadStore (P "netflix.com" "netflixminer")
assume PAx17:       CanReadLocalFile "moviegenres.txt"

val get_nth: list<'a> -> int -> 'a
let get_nth l n = match l with
    | Cons h t -> if n = 0 then h else get_nth t (n - 1)

val get_id: xelem -> string
let get_id e =
    let href = GetAttribute e "href" in
    let sp1 = get_nth (Split href "&") 3 in
    let sp2 = get_nth (Split sp1 "=") 1 in
    let sp3 = get_nth (Split sp2 "_") 0 in
        get_nth (Split sp3 "&") 3

val make_link: string -> string
let make_link x = StringConcat "b0" (StringConcat x "_0")

val submitHandler:  DOMEvent ->
                    unit
let submitHandler ev = match ev.doc.dhost with
    | "netflix.com" ->
        let flixprov = PCons (P "netflix.com" "netflixminer") PNil in
        let target = GetEvTarget flixprov "netflixminer" ev in
        let movieId = bind flixprov target get_id in
        let linkId = bind flixprov movieId make_link in
        let titleEl = GetElementByTrackedId flixprov "netflixminer" ev.doc linkId in
        let movieTitle = bind flixprov titleEl (fun x -> GetAttribute x "href") in
        let cats = bind flixprov movieTitle (fun x -> Cons "Top/Entertainment/Movies" Nil) in
            AddEntry flixprov movieTitle "movie" cats (let_mutate ())

val attachToRatings: ({d:DocHandle | d.dhost = "netflix.com"}) ->
                     ({s:string | CanReadDOMClass "netflix.com" s}) ->
                     unit
let attachToRatings doc cl =
    let flixprov = PCons (P "netflix.com" "netflixminer") PNil in
    let els = GetElementsByClass flixprov "netflixminer" doc cl in
    let addEvListen = fun x -> AddEventListener flixprov x "onclick" submitHandler in
        iterate addEvListen els

val docLoadHandler: ({d:DocHandle | CanHandleSites d.dhost}) ->
                    unit
let docLoadHandler doc = match doc.dhost with | "netflix.com" ->
    let rv1 = attachToRatings doc "rv1" in
```

```
67      let rv1 = attachToRatings doc "rv2" in
68      let rv1 = attachToRatings doc "rv3" in
69      let rv1 = attachToRatings doc "rv4" in
70      let rv1 = attachToRatings doc "rv5" in
71          ()
72
73  val mymap: list<'a> -> ('a -> 'b) -> list<'b>
74  let mymap l f = match l with
75      | Cons h t -> Cons (f h) (mymap t f)
76      | Nil -> Nil
77
78  val isGenre: string -> string -> list<(string * string)> -> bool
79  let isGenre title genre gl = match gl with
80      | Cons (ctitle, cgenre) t -> if title = ctitle && genre = cgenre then true else isGenre title genre gl
81      | Nil -> false
82
83  val filterByList: list<string> -> list<(string * string)> -> string -> list<string>
84  let filterByList movies gl genre = match movies with
85      | Cons h t -> if (isGenre h genre gl) then (Cons h (filterByList t gl genre)) else (filterByList t gl genre)
86      | Nil -> Nil
87
88  val filterByGenre: string -> list<string> -> list<string>
89  let filterByGenre genre movies =
90      let movieGenres = ReadFileLines "moviegenres.txt" in
91      let genreList = mymap movieGenres (fun x -> match (Split x ":") with | Cons m (Cons g Nil) -> (m, g)) in
92          filterByList movies genreList genre
93
94  val doGetMovies:    string ->
95                      ({s:string | CanServeInformation s (P "netflix.com" "netflixminer")}) ->
96                      unit
97  let doGetMovies genre cdom =
98      let myprov = PCons (P "netflix.com" "netflixminer") PNil in
99      let flixEnts = GetStoreEntriesByTopic myprov "movie" in
100     let genreFlix = bind myprov flixEnts (filterByGenre genre) in
101         ExtensionReturn cdom myprov genreFlix
102
103 val getMoviesByGenre:   string -> string -> unit
104 let getMoviesByGenre genre cdom = match cdom with
105     | "fandango.com" -> doGetMovies genre cdom
106     | "amazon.com" -> doGetMovies genre cdom
107     | "metacritic.com" -> doGetMovies genre cdom
108
109 val main: mut_capability -> unit
110 let main mcap =
111     let d = AddNewDocHandler "netflix.com" docLoadHandler in
112     InstallExternalInterface getMoviesByGenre
```

A.2 Twitter Miner

```
// TwitterMiner - Fine version
module TwitterMiner

open Url
open RePrivPolicy
open RePrivAPI

// Policy assumptions

assume extid: ExtensionId "twitterminer"
assume PAx1: CanCommunicateXHR "twitter.com"
assume PAx2: forall (s:string) . (ExtensionId s) => CanUpdateStore (P "twitter.com" s)

// The actual miner

val GetDescription: xdoc -> string
let GetDescription d =
    let allMsgs = ReadXDocEls d "item" (fun x -> true) "description" in
    match allMsgs with
        | Cons h t -> h
        | Nil -> ""

val CollectLatestFeed:  ({s:string | ExtensionId s}) ->
                        unit ->
                        unit
let CollectLatestFeed extid u =
    let twitterProv = (simple_prov "twitter.com" extid) in
    let reqUrl = (mkUrl "http" "twitter.com" "statuses/user_timeline/19852608.rss") in
    let twitFeed = MakeXDocRequest reqUrl extid twitterProv (let_mutate ()) in
    let currentMsg = (bind twitterProv twitFeed GetDescription) in
    let categories = (bind twitterProv currentMsg ClassifyText) in
        AddEntry twitterProv currentMsg "tweet" categories (let_mutate ())

val main: unit -> unit
let main x =
    SetTimeout 60000 (CollectLatestFeed "twitterminer")
```

A.3 Bing Miner

```
// BingMiner - Fine version
module BingMiner

open RePrivPolicy
open RePrivAPI

// Policy assumptions

assume extid: ExtensionId "bingminer"
assume PAx1: forall (s:string) . (ExtensionId s) => CanUpdateStore (P "bing.com" s)
assume PAx2: CanCaptureEvents "submit" (P "bing.com" "bingminer")
assume PAx3: CanReadDOMId "bing.com" "sb_form"
assume PAx4: CanHandleSites "bing.com"

val submitHandler:  DOMEvent ->
                    unit
let submitHandler ev = match ev.doc.dhost with
    | "bing.com" ->
        let bingprov = PCons (P "bing.com" "bingminer") PNil in
        let searchEl = GetElementById bingprov "bingminer" ev.doc "sb_form" in
        let searchVal = bind bingprov searchEl (fun x -> GetElValue x) in
        let categories = bind bingprov searchVal ClassifyText in
            AddEntry bingprov searchVal "search" categories (let_mutate ())
    | _ -> ()

val docLoadHandler: ({d:DocHandle | CanHandleSites d.dhost}) ->
                    unit
let docLoadHandler doc = match doc.dhost with | "bing.com" ->
    let bingprov = PCons (P "bing.com" "bingminer") PNil in
    let el = GetElementById bingprov "bingminer" doc "sb_form" in
        AddEventListener bingprov el "submit" submitHandler

val main: mut_capability -> unit
let main mcap =
    AddNewDocHandler "bing.com" docLoadHandler
```

A.4 Glue Miner

```
// GlueMiner - Fine code
module GlueMiner

open RePrivPolicy
open RePrivAPI
open Url

// Policy assumptions

assume extid:   ExtensionId "glueminer"
assume PAx1:    CanCommunicateXHRTracked "getglue.com" (P "netflix.com" "netflixminer")
assume PAx2:    CanCommunicateXHRTracked "getglue.com" (P "twitter.com" "twitterminer")
assume PAx3:    CanCommunicateXHRTracked "getglue.com" (P "facebook.com" "facebookminer")
assume PAx4:    CanReadStore (P "netflix.com" "netflixminer")
assume PAx5:    CanReadStore (P "twitter.com" "twitterminer")
assume PAx6:    CanReadStore (P "facebook.com" "facebookminer")
assume PAx7:    CanServeInformation "fandango.com" (P "getglue.com" "glueminer")
assume PAx8:    CanServeInformation "fandango.com" (P "netflix.com" "netflixminer")
assume PAx9:    CanServeInformation "linkedin.com" (P "getglue.com" "glueminer")
assume PAx10:   CanServeInformation "linkedin.com" (P "twitter.com" "twitterminer")
assume PAx11:   CanServeInformation "linkedin.com" (P "facebook.com" "facebookminer")

val concat_strs: list<string> -> string -> string
let concat_strs l a = match l with
    | Cons h t -> concat_strs t (StringConcat a h)
    | Nil -> a

val movieQueryString: int -> list<string> -> string
let movieQueryString n l =
    if n = 1 then match l with
        | Cons h t -> StringConcat "/v2/object/links?objectId=movies/" h
        | Nil -> "/v2/object/links?objectId="
    else match l with
        | Cons h t -> movieQueryString (n - 1) t
        | Nil -> "/v2/object/links?objectId="

val twitQueryString: int -> list<string> -> string
let twitQueryString n l =
    if n = 1 then match l with
        | Cons h t -> StringConcat "/v2/object/findObjects?q=" (concat_strs (ClassifyText h) "")
        | Nil -> "/v2/object/findObjects?q="
    else match l with
        | Cons h t -> twitQueryString (n - 1) t
        | Nil -> "/v2/object/findObjects?q="

val fbQueryString: int -> list<string> -> string
let fbQueryString n l =
    if n = 1 then match l with
        | Cons h t -> StringConcat "/v2/object/links?objectId=" h
        | Nil -> "/v2/object/links?objectId="
    else match l with
        | Cons h t -> fbQueryString (n - 1) t
        | Nil -> "/v2/object/links?objectId="


val getResult:  p:({p:provs | AllCanCommunicateXHRTracked "getglue.com" p}) ->
                tracked<list<string>,p> ->
                bothprov:({b:provs | forall (pr:prov) . InProvs pr b <=>
                  (InProvs pr p || pr = (P "getglue.com" "glueminer"))}) ->
                (int -> list<string> -> string) ->
                tracked<string,bothprov>
let getResult p t both f =
    let qstr1 = bind p t (f 1) in
    let qstr2 = bind p t (f 2) in
    let qstr3 = bind p t (f 3) in
    let result1 = MakeRequestTracked p "getglue.com" qstr1 "glueminer" both (let_mutate ()) in
    let result2 = MakeRequestTracked p "getglue.com" qstr2 "glueminer" both (let_mutate ()) in
    let result3 = MakeRequestTracked p "getglue.com" qstr3 "glueminer" both (let_mutate ()) in
    let cat1 = bind2 both result1 both result2 StringConcat both in
        bind2 both cat1 both result3 StringConcat both

val socialResults:  string ->
                    ({s:string | s = "linkedin.com"}) ->
                    unit
let socialResults topic cdom =
    let twitprov = PCons (P "twitter.com" "twitterminer") PNil in
    let gluetwitprov = PCons (P "getglue.com" "glueminer") twitprov in
    let fbprov = PCons (P "facebook.com" "facebookminer") PNil in
    let gluefbprov = PCons (P "getglue.com" "glueminer") fbprov in
```

```
80      let allprovs = PCons (P "twitter.com" "twitterminer") gluefbprov in
81      let twitEnts = GetStoreEntriesByTopic twitprov "tweet" in
82      let fbEnts = GetStoreEntriesByTopic fbprov "like" in
83      let twitResponse = getResult twitprov twitEnts gluetwitprov twitQueryString in
84      let fbResponse = getResult fbprov fbEnts gluefbprov fbQueryString in
85      let finalresult = bind2 gluetwitprov twitResponse gluefbprov fbResponse StringConcat allprovs in
86          ExtensionReturn cdom allprovs finalresult
87
88  val movieResults:   string ->
89                      ({s:string | s = "fandango.com"}) ->
90                      unit
91  let movieResults topic cdom =
92      let flixprov = PCons (P "netflix.com" "netflixminer") PNil in
93      let bothprov = PCons (P "getglue.com" "glueminer") flixprov in
94      let flixEnts = GetStoreEntriesByTopic flixprov "movie" in
95          ExtensionReturn cdom bothprov (getResult flixprov flixEnts bothprov movieQueryString)
96
97  val resultsByTopic: string -> string -> unit
98  let resultsByTopic topic cdom = match cdom with
99      | "fandango.com" -> movieResults topic cdom
100     | "linkedin.com" -> socialResults topic cdom
101
102 val main: unit -> unit
103 let main x = InstallExternalInterface resultsByTopic
```

B. Case Studies

B.1 News Personalizer

```
module NewsPersonalizer

open FineAPI
open Url

assume extid:   ExtensionId "newspers"
assume PAx1:    CanHandleSites "mobile.nytimes.com"
assume PAx3:    CanCommunicateDOM (P "services.digg.com" "newspers") "mobile.nytimes.com"
assume PAx4:    CanCommunicateDOM (P "repriv" "repriv") "mobile.nytimes.com"
assume PAx5:    forall (e:elt) . CanReadAttr e "class"
assume PAx6:    CanReadDOMId "mobile.nytimes.com" "container"
assume PAx7:    CanCommunicateXHRTracked "services.digg.com" (P "repriv" "repriv")

val nthdata:
    list<'a> ->
    int ->
    'a
let nthdata data i =
    match data with
        | Cons h1 t1 -> if not (i = 0) then nthdata t1 (i - 1) else h1

val nthdataT:
    p:provs ->
    tracked<list<'a>,p> ->
    int ->
    tracked<'a,p>
let nthdataT p data i =
    match data with
        | Tag d p -> Tag (nthdata d i) p

val setMinorHeadline:
    ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
      p1 = (P "repriv" "repriv") && p2 = (P "services.digg.com" "newspers")}) ->
    ({p:provs | exists (pr:prov) . InProvs pr p && pr = (P "mobile.nytimes.com" "newspers")}) ->
    tracked<string, bp> ->
    tracked<string, bp> ->
    tracked<elt, p> ->
    mut_capability ->
    mut_capability
let setMinorHeadline bp p line link el cap =
    let elc = getChildrenT p el in
    let headl = bind p elc (fun x -> nthdata x 0) in
    let (_, cap1) = SetTextContent p "mobile.nytimes.com" bp headl line cap in
    let (_, cap2) = setAttrT p "mobile.nytimes.com" bp headl "href" link cap1 in
    cap2

val setTopHeadline:
    ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
      p1 = (P "repriv" "repriv") && p2 = (P "services.digg.com" "newspers")}) ->
    ({p:provs | exists (pr:prov) . InProvs pr p && pr = (P "mobile.nytimes.com" "newspers")}) ->
    tracked<string, bp> ->
    tracked<string, bp> ->
    tracked<string, bp> ->
    tracked<elt, p> ->
    mut_capability ->
    mut_capability
let setTopHeadline bp p line link sum el cap =
    let elc = getChildrenT p el in
    let headl = bind p elc (fun x -> nthdata x 0) in
    let summary = bind p elc (fun x -> nthdata x 1) in
    let (_, cap1) = SetTextContent p "mobile.nytimes.com" bp headl line cap in
    let (_, cap2) = setAttrT p "mobile.nytimes.com" bp headl "href" link cap1 in
    let (_, cap3) = SetTextContent p "mobile.nytimes.com" bp summary sum cap2 in
    cap3

val findNews:
  list<elt> ->
    elt
let findNews lst =
    match lst with
        | Cons h t -> if (getAttr h "class") = "eg␣sp" then h else findNews t

// Maps a repriv taxonomy topic to a digg.com news topic
val mapTopic:
  string ->
  string
let mapTopic x = // removed for brevity

```

```
val getstories:
  ({p:provs | Singleton p && (exists (pp:prov) . pp = (P "repriv" "repriv") && InProvs pp p)}) ->
  tracked<list<string>,p> ->
  ({p2:provs | exists (pr:prov) . InProvs pr p2 && pr = (P "services.digg.com" "newspers")}) ->
  ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
    p1 = (P "repriv" "repriv") && p2 = (P "services.digg.com" "newspers")}) ->
  mut_capability ->
  ((tracked<list<string>, bp> * tracked<list<string>, bp> * tracked<list<string>, bp>) * mut_capability)
let getstories p topics p2 bp cap =
  let q = bind p topics (fun x -> match x with
                                  | Cons h t ->
                                  StringConcat "http://services.digg.com/..." (mapTopic h)) in
  let (xstr, cap1) = MakeTrackedXDocRequestT p "services.digg.com" q "newspers" bp cap in
  let readtitle = fun x -> readXDocEls x "story" (fun x -> true) "title" in
  let readlink = fun x -> readXDocEls x "story" (fun x -> true) "link" in
  let readabstract = fun x -> readXDocEls x "story" (fun x -> true) "description" in
  (((bind bp xstr readtitle), (bind bp xstr readlink), (bind bp xstr readabstract)), cap1)

val docLoadHandler:
  doc ->
  mut_capability ->
  unit
let docLoadHandler d cap =
    let p1 = simple_prov "mobile.nytimes.com" "newspers" in
    let p2 = simple_prov "services.digg.com" "newspers" in
    let rp = simple_prov "repriv" "repriv" in
    let bp = comp_prov rp p2 in
    let topi = getTopInterests rp 1 in
    let ((headlines, urls, summaries), cap1) = getstories rp topi p2 bp cap in
    match (getDocHost d) with | ("mobile.nytimes.com", dr) ->
    let cont = getEltByIdT p1 "newspers" dr "container" in
    let cc = getChildrenT p1 cont in
    let storyCont = bind p1 cc findNews in
    let storyChildren = getChildrenT p1 storyCont in
    let cap2 = setTopHeadline bp p1 (nthdataT bp headlines 0)
                                    (nthdataT bp urls 0)
                                    (nthdataT bp summaries 0)
                                    (nthdataT p1 storyChildren 0) cap1 in
    let cap3 = setMinorHeadline bp p1 (nthdataT bp headlines 0)
                                      (nthdataT bp urls 0)
                                      (nthdataT p1 storyChildren 0) cap2 in
    ()

val main: mut_capability -> unit
let main cap =
  AddNewDocHandler "mobile.nytimes.com" (fun x -> docLoadHandler x cap)
```

B.2 Search Personalizer

```
module SearchPersonalizer

open FineAPI
open Url

assume extid:   ExtensionId "searchpers"
assume PAx1:    CanCommunicateXHRTracked "boss.yahooapis.com" (P "m.bing.com" "searchpers")
assume PAx2:    CanReadDOMId "m.bing.com" "Q"
assume PAx3:    CanHandleSites "m.bing.com"
assume PAx4:    forall (e:elt) . CanReadValue e
assume PAx5:    CanReadStore (P "m.bing.com" "searchpers")
assume PAx6:    CanReadDOMClass "m.bing.com" "s15"
assume PAx7:    forall (e:elt) . CanReadAttr e "class"
assume PAx8:    CanCommunicateDOM (P "m.bing.com" "searchpers") "m.bing.com"
assume PAx9:    CanCommunicateDOM (P "boss.yahooapis.com" "searchpers") "m.bing.com"
assume PAx10:   CanCaptureEvents "submit" (P "m.bing.com" "searchpers")
assume PAx11:   CanUpdateStore (P "m.bing.com" "searchpers")
assume PAx12:   CanCaptureEvents "click" (P "m.bing.com" "searchpers")

val yahooProv: provs
let yahooProv = PCons (P "boss.yahooapis.com" "searchpers") PNil

val bingProv: provs
let bingProv = PCons (P "m.bing.com" "searchpers") PNil

val inlist:
    'a ->
    list<'a> ->
    bool
let inlist el lst =
    match lst with
        | Cons hd tl -> if hd = el then true else (inlist el tl)
        | Nil -> false

val allexcept:
    list<string> ->
    list<string> ->
    list<string>
let allexcept primary exclude =
    match primary with
        | Cons hd tl -> if (inlist hd exclude) then (allexcept tl exclude) else (Cons hd (allexcept tl exclude))
        | Nil -> Nil

val append:
    list<'a> ->
    list<'a> ->
    list<'a>
let append l1 l2 =
    match l1 with
        | Cons h t -> Cons h (append l2 t)
        | Nil -> l2

// Queries the BOSS API
val runsearch:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
    tracked<string,ip> ->
    ({p:provs | exists (pr:prov) . InProvs pr p && pr = (P "boss.yahooapis.com" "searchpers")}) ->
  ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
    p1 = (P "boss.yahooapis.com" "searchpers") && p2 = (P "m.bing.com" "searchpers")}) ->
  mut_capability ->
    ((tracked<list<string>, bp> * tracked<list<string>, bp> * tracked<list<string>, bp>) * mut_capability)
let runsearch ip query p bp cap =
  let tq1 = bind ip query (fun x -> StringConcat x "?appid={appid}&format=xml&count=50") in
  let q = bind ip tq1 (fun x -> StringConcat "ysearch/web/v1" x) in
    let (xstr, cap1) = MakeTrackedXDocRequestT ip "boss.yahooapis.com" q "searchpers" bp cap in
    let readclos = fun x -> readXDocEls x "{http://www.inktomi.com/}result" (fun x -> true)
      "{http://www.inktomi.com/}url" in
    let readtitle = fun x -> readXDocEls x "{http://www.inktomi.com/}result" (fun x -> true)
      "{http://www.inktomi.com/}title" in
    let readabstract = fun x -> readXDocEls x "{http://www.inktomi.com/}result" (fun x -> true)
      "{http://www.inktomi.com/}abstract" in
    (((bind bp xstr readclos), (bind bp xstr readtitle), (bind bp xstr readabstract)), cap1)

// Return results previously selected by user
val lastselected:
    list<string> ->
    list<string> ->
    list<string> ->
    list<string>
```

```
let lastselected searchres prev ret =
    match searchres with
        | Cons hd tl -> if (inlist hd prev) then (lastselected tl prev (Cons hd ret)) else (lastselected tl prev ret)
        | Nil -> ret

// Sort the first argument by frequency of host visitation
val reorder:
    string ->
    int ->
    list<(string * int)> ->
    list<(string * int)>
let reorder ent freq lst =
    match lst with
        | Cons (s, n) tl -> if freq = n then (Cons (ent, freq) lst) else (Cons (s, n) (reorder ent freq tl))
        | Nil -> Nil

val count:
  string ->
  list<string> ->
  int
let count el l = match l with
  | Cons el tl -> 1 + (count el tl)
  | Cons _ tl -> count el tl
  | Nil -> 0

val counthostvisits:
  ({p:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 p}) ->
  tracked<list<string>,p> ->
  tracked<list<(string * int)>, p>
let counthostvisits p l =
  let all = GetStoreEntriesByTopic p "searchresult" in
  let first = bind p l (fun x -> match x with | Cons h t -> h) in
  let rest = bind p l (fun x -> match x with | Cons h t -> t | Nil -> Nil) in
  let num = bind2 p first p all count p in
  let pair = bind2 p first p num (fun x y -> (x, y)) p in
  bind2 p pair p (counthostvisits p rest) (fun x y -> Cons x y) p

val countvisits:
  list<(string * int)> ->
    string ->
    int
let countvisits freqs ent = match freqs with
  | Cons (ent, i) tl -> i
  | Cons _ tl -> countvisits tl ent
  | nil -> 0

val hostfrequency:
    list<string> ->
  list<(string * int)> ->
    list<(string * int)> ->
    list<string>
let hostfrequency searchres freqs ret =
    match searchres with
        | Cons hd tl -> hostfrequency tl freqs (reorder hd (countvisits freqs hd) ret)
        | Nil -> map ret (fun x -> match x with | (x, y) -> x)

// The actual re-ranking function
val filterBySearch:
    string ->
    list<string> ->
    list<string>
let filterBySearch q l =
    match l with
        | Cons h t -> if (Split h ":" 0) = q then Cons (Split h ":" 1) (filterBySearch q t) else (filterBySearch q t)
        | Nil -> Nil

val getselected:
  bp:provs ->
    tracked<string,bp> ->
  ({p:provs | AllCanReadStore p}) ->
  ({fp:provs | forall (tp:prov) . (InProvs tp p || InProvs tp bp) <=> InProvs tp fp}) ->
    tracked<list<string>, fp>
let getselected bp q p fp =
    let all = GetStoreEntriesByTopic p "searchresult" in
    bind2 bp q p all filterBySearch fp

val rerank:
    ({p:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 p}) ->
    tracked<string, p> ->
    ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
      p1 = (P "boss.yahooapis.com" "searchpers") && p2 = (P "m.bing.com" "searchpers")}) ->
```

```
  mut_capability ->
  ((tracked<list<string>,bp> * (tracked<list<string>,bp> * tracked<list<string>,bp> * tracked<list<string>,bp>)) *
      mut_capability)
let rerank p t bp cap =
  let p1 = simple_prov "boss.yahooapis.com" "searchpers" in
  let (defltall, cap1) = runsearch p t p1 bp cap in
  match defltall with | (deflt, _, _) ->
  let prev = getselected p t p p in
  let reorder1 = bind2 bp deflt p prev (fun x y -> lastselected x y Nil) bp in
  let pruned1 = bind2 bp deflt bp reorder1 allexcept bp in
  let hostvisits = counthostvisits bp pruned1 in
  let reorder2 = bind2 bp pruned1 bp hostvisits (fun x y -> hostfrequency x y Nil) bp in
  let pruned2 = bind2 bp pruned1 bp reorder2 allexcept bp in
  let append1 = bind2 bp reorder1 bp reorder2 append bp in
    ((bind2 bp append1 bp pruned2 append bp, defltall), cap1)

// Browser interfacing stuff
val findLink:
  list<elt> ->
  elt
let findLink lst =
    match lst with
        | Cons h t -> if (getAttr h "class") = "Link" then h else findLink t

val findDispUrl:
  list<elt> ->
  elt
let findDispUrl lst =
    match lst with
        | Cons h t -> if (getAttr h "class") = "c2" then h else findLink t

val findSpan:
  list<elt> ->
  elt
let findSpan lst =
    match lst with
        | Cons h t -> if (getElType h) = "span" then h else findSpan t

val changeLink:
  ({p:provs | forall (p1:prov) . (InProvs p1 p) <=> (p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<elt,p> ->
  ({bp:provs | forall (p1:prov) . InProvs p1 bp <=>
    (p1 = (P "boss.yahooapis.com" "searchpers") || p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<string,bp> ->
  tracked<string,bp> ->
  mut_capability ->
  (unit * mut_capability)
let changeLink p el bp url title cap =
  let t1 = getChildrenT p el in
  let linkEl = bind p t1 findLink in
  let (t2, cap1) = setAttrT p "m.bing.com" bp linkEl "href" url cap in
  let (t3, cap2) = SetTextContent p "m.bing.com" bp linkEl title cap1 in
  ((), cap2)

val changeDispUrl:
  ({p:provs | forall (p1:prov) . (InProvs p1 p) <=> (p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<elt,p> ->
  ({bp:provs | forall (p1:prov) . InProvs p1 bp <=>
    (p1 = (P "boss.yahooapis.com" "searchpers") || p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<string,bp> ->
  mut_capability ->
  (unit * mut_capability)
let changeDispUrl p el bp url cap =
  let t1 = getChildrenT p el in
  let dispEl = bind p t1 findDispUrl in
  let (t3, cap1) = SetTextContent p "m.bing.com" bp dispEl url cap in
  ((), cap1)

val changeDescription:
  ({p:provs | forall (p1:prov) . (InProvs p1 p) <=> (p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<elt,p> ->
  ({bp:provs | forall (p1:prov) . InProvs p1 bp <=>
    (p1 = (P "boss.yahooapis.com" "searchpers") || p1 = (P "m.bing.com" "searchpers"))}) ->
  tracked<string,bp> ->
  mut_capability ->
  (unit * mut_capability)
let changeDescription p el bp desc cap =
  let t1 = getChildrenT p el in
  let spanEl = bind p t1 findSpan in
  let (t2, cap1) = SetTextContent p "m.bing.com" bp spanEl desc cap in
  ((), cap1)
```

```
val modifySearch:
  ({p:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 p}) ->
  tracked<elt,p> ->
  ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
    p1 = (P "boss.yahooapis.com" "searchpers") && p2 = (P "m.bing.com" "searchpers")}) ->
  tracked<string,bp> ->
  tracked<string,bp> ->
  tracked<string,bp> ->
  mut_capability ->
  (unit * mut_capability)
let modifySearch p el bp url title abs cap =
  let children = getChildrenT p el in
  let (t1, cap1) = changeLink p el bp url title cap in
  let (t2, cap2) = changeDescription p el bp abs cap1 in
  let (t3, cap3) = changeDispUrl p el bp url cap2 in
  ((), cap3)

val submitHandler:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
  ({p:provs | exists (pr:prov) . InProvs pr p && pr = (P "boss.yahooapis.com" "searchpers")}) ->
  ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
    p1 = (P "boss.yahooapis.com" "searchpers") && p2 = (P "m.bing.com" "searchpers")}) ->
  evt ->
  mut_capability ->
  unit
let submitHandler ip p bp e cap =
  let d = getEvTarget e in
  match (getDocHost d) with | ("m.bing.com", dr) ->
  let resDiv = getEltByIdT ip "searchpers" dr "Q" in
  let q = bind ip resDiv getElValue in
  let t = MakeTrackedXDocRequestT ip "boss.yahooapis.com" q "searchpers" bp cap in
  ()

val selectionHandler:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
  tracked<string,ip> ->
  tracked<string,ip> ->
  evt ->
  mut_capability ->
  unit
let selectionHandler p q url e cap =
  let tconcat = bind2 p q p url StringConcat p in
  let tnil = bind p q (fun x -> Nil) in
  let t = AddEntry p tconcat "searchresult" tnil cap in
  ()

val hookSel:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
  tracked<elt,ip> ->
  tracked<string,ip> ->
  mut_capability ->
  (unit * mut_capability)
let hookSel p el q cap =
  let t1 = getChildrenT p el in
  let linkEl = bind p t1 findLink in
  let url = getAttrT p el "href" in
  let (t2, cap1) = addListenerWithCap p el "click" (selectionHandler p q url) cap in
  ((), cap1)

val hookSelections:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
  tracked<string,ip> ->
  doc ->
  mut_capability ->
  (unit * mut_capability)
let hookSelections p q d cap =
  match (getDocHost d) with | ("m.bing.com", dr) ->
  let reselts = getEltsByClassNameT p "searchpers" dr "s15" in
  let (t1, cap1) = hookSel p (nthdata reselts 0) q cap in
  let (t2, cap2) = hookSel p (nthdata reselts 1) q cap1 in
  let (t3, cap3) = hookSel p (nthdata reselts 2) q cap2 in
  let (t4, cap4) = hookSel p (nthdata reselts 3) q cap3 in
  let (t5, cap5) = hookSel p (nthdata reselts 4) q cap4 in
  let (t6, cap6) = hookSel p (nthdata reselts 5) q cap5 in
  let (t7, cap7) = hookSel p (nthdata reselts 6) q cap6 in
  let (t8, cap8) = hookSel p (nthdata reselts 7) q cap7 in
  let (t9, cap9) = hookSel p (nthdata reselts 8) q cap8 in
  let (t10, cap10) = hookSel p (nthdata reselts 9) q cap9 in
  ((), cap10)
```

```
val nthdata:
    list<'a> ->
    int ->
    'a
let nthdata data i =
    match data with
        | Cons h1 t1 -> if not (i = 0) then nthdata t1 (i - 1) else h1

val updateResults:
  ({ip:provs | exists (p1:prov) . p1 = (P "m.bing.com" "searchpers") && InProvs p1 ip}) ->
  ({p:provs | exists (pr:prov) . InProvs pr p && pr = (P "boss.yahooapis.com" "searchpers")}) ->
  ({bp:provs | exists (p1:prov), (p2:prov) . InProvs p1 bp && InProvs p2 bp &&
    p1 = (P "boss.yahooapis.com" "searchpers") && p2 = (P "m.bing.com" "searchpers")}) ->
  int ->
  list<tracked<elt, ip>> ->
  tracked<list<string>, bp> ->
  tracked<list<string>, bp> ->
  tracked<list<string>, bp> ->
  mut_capability ->
  (unit * mut_capability)
let updateResults p2 p1 bp i reselts c1 c2 c3 cap =
  let data1 = (bind bp c1 (fun x -> nthdata x i),
               bind bp c2 (fun x -> nthdata x i),
               bind bp c3 (fun x -> nthdata x i)) in
  match data1 with | (uc1, uc2, uc3) ->
  let (_, cap1) = modifySearch p2 (nthdata reselts i) bp uc1 uc2 uc3 cap in
  ((), cap1)

val docLoadHandler:
  doc ->
  mut_capability ->
  unit
let docLoadHandler d cap =
  let p1 = simple_prov "boss.yahooapis.com" "searchpers" in
  let p2 = simple_prov "m.bing.com" "searchpers" in
  let bp = comp_prov p1 p2 in
  match (getDocHost d) with | ("m.bing.com", dr) ->
  let resDiv = getEltByIdT p2 "searchpers" dr "Q" in
  let q = bind p2 resDiv getElValue in
  let (t1, cap1) = rerank p2 q bp cap in
  let (t2, cap2) = addListenerWithCap p2 resDiv "submit" (submitHandler p2 p1 bp) cap1 in
  match t1 with | (reranked, allinfo) ->
  let reselts = getEltsByClassNameT p2 "searchpers" dr "s15" in
  match allinfo with | (c1, c2, c3) ->
  let (_, cap3) = updateResults p2 p1 bp 0 reselts c1 c2 c3 cap2 in
  let (_, cap4) = updateResults p2 p1 bp 1 reselts c1 c2 c3 cap3 in
  let (_, cap5) = updateResults p2 p1 bp 2 reselts c1 c2 c3 cap4 in
  let (_, cap6) = updateResults p2 p1 bp 3 reselts c1 c2 c3 cap5 in
  let (_, cap7) = updateResults p2 p1 bp 4 reselts c1 c2 c3 cap6 in
  let (_, cap8) = updateResults p2 p1 bp 5 reselts c1 c2 c3 cap7 in
  let (_, cap9) = updateResults p2 p1 bp 6 reselts c1 c2 c3 cap8 in
  let (_, cap10) = updateResults p2 p1 bp 7 reselts c1 c2 c3 cap9 in
  let (_, cap11) = updateResults p2 p1 bp 8 reselts c1 c2 c3 cap10 in
  let (_, cap12) = updateResults p2 p1 bp 9 reselts c1 c2 c3 cap11 in
  let (_, cap13) = hookSelections p2 q d cap12 in
  ()

val main: mut_capability -> unit
let main cap =
  AddNewDocHandler "m.bing.com" (fun x -> docLoadHandler x cap)
```

C. RePriv API (representative subset)

```
// Policy definitions

private type prov =
    | P : string -> string -> prov

private type provs :: * =
    | PNil : provs
    | PCons : prov -> provs -> provs

prop InProvs :: prov => provs => *
assume HdProvs:forall (p:prov), (tl:provs). InProvs p (PCons p tl)
assume TlProvs:forall (p:prov), (q:prov), (tl:provs).
    InProvs p tl => InProvs p (PCons q tl)
assume notinPNil: forall (p:prov). not (InProvs p PNil)
assume notinPCons: forall (a:prov), (b:prov), (tl:provs).
    ((not (InProvs a tl)) && (not (a=b))) => not (InProvs a (PCons b tl))

prop Singleton :: provs => *
assume provsSing : forall (ps:provs). (exists (p:prov). (ps = (Cons p Nil))) <=> (Singleton ps)

type tracked :: * => provs => * =
    | Tag : 'a -> p:provs -> tracked<'a,p>

val simple_prov:    d:string ->
                    e:string ->
                    ({p:provs | In (P d e) p && Singleton p})
let simple_prov d e = Cons (P d e) Nil

val comp_prov:  p1:provs ->
                p2:provs ->
                ({p:provs | forall (pr:prov) .
                  ((InProvs pr p1) || (InProvs pr p2)) <=> (InProvs pr p)})
let rec comp_prov p1 p2 = match p1 with
    | PCons p tl -> PCons p (comp_prov tl p2)
    | PNil -> p2

val bind: p:provs -> tracked<'a,p> -> ('a -> 'b) -> tracked<'b,p>
let bind p v f = match v with
    | Tag d p -> Tag (f d) p

val bind2:  p1:provs ->
            tracked<'a,p1> ->
            p2:provs ->
            tracked<'b,p2> ->
            ('a -> 'b -> 'c) ->
            p3:({p3:provs | forall (pr:prov) . ((InProvs pr p1) || (InProvs pr p2)) <=> (InProvs pr p3)}) ->
            tracked<'c,p3>
let bind2 p1 t1 p2 t2 f p3 =
    match t1 with | Tag d1 p1 ->
    match t2 with | Tag d2 p2 ->
        Tag (f d1 d2) p3

private type mut_capability :: + =
    | MCap : mut_capability

val let_mutate: unit -> mut_capability
let let_mutate x = MCap

// CanReadStore :: source prov
prop CanReadStore :: prov => *

// CanUpdateStore :: source prov
prop CanUpdateStore :: prov => *

// AllCanUpdateStore :: source provs
prop AllCanUpdateStore :: provs => *
assume AllCanUp: forall (ps:provs) . (forall (p:prov) .
    (InProvs p ps) => (CanUpdateStore p)) <=> AllCanUpdateStore ps

// AllCanReadStore :: source provs
prop AllCanReadStore :: provs => *
assume AllCanRead: forall (ps:provs) . (forall (p:prov) .
    (InProvs p ps) => (CanReadStore p)) <=> AllCanReadStore ps

// CanServeInformation :: dest host => source prov
prop CanServeInformation :: string => prov => *

// AllCanServeInformation :: dest host => source provs
prop AllCanServeInformation :: string => provs => *
```

```
assume AllCanServe: forall (ps:provs), (s:string) . (forall (p:prov) .
    (InProvs p ps) => (CanServeInformation s p)) <=> AllCanServeInformation s ps

// ExtensionId :: extension id
prop ExtensionId :: string => *

// CanCommunicateXHR :: dest host
prop CanCommunicateXHR :: string => *

// CanCommunicateXHRTracked :: dest host => source prov
prop CanCommunicateXHRTracked :: string => prov => *

// AllCanCommunicateXHRTracked :: dest host => source provs
prop AllCanCommunicateXHRTracked :: string => provs => *
assume AllCanCom: forall (h:string), (ps:provs), (p:prov) .
    ((InProvs p ps) => (CanCommunicateXHRTracked h p)) <=> AllCanCommunicateXHRTracked h ps

// CanCommunicateDOM :: source prov => host
prop CanCommunicateDOM :: prov => string => *

// AllCanCommunicateDOM :: source provs => host => *
prop AllCanCommunicateDOM :: provs => string => *
assume AllCanCommDOM: forall (ps:provs), (s:string) . (forall (p:prov) .
    (InProvs p ps) => (CanCommunicateDOM p s)) <=> AllCanCommunicateDOM ps s

// CanReadLocalFile :: file name
prop CanReadLocalFile :: string => *

// CanCaptureEvents :: element type => element provenance
prop CanCaptureEvents :: string => prov => *

// AllCanCapture :: element type => element provs
prop AllCanCapture :: string => provs => *
assume AllCanCap: forall (ps:provs), (s:string) . (forall (p:prov) .
    (InProvs p ps) => (CanCaptureEvents s p)) <=> AllCanCapture s ps

// CanReadDOMId :: page host => id name
prop CanReadDOMId :: string => string => *

// CanReadDOMClass :: page host => class name
prop CanReadDOMClass :: string => string => *

// CanHandleSites :: site host
prop CanHandleSites :: string => *

// EltHost :: element => host
prop EltHost :: elt => string => *

//
// API Wrappers
//

extern API val getDocHost:
    d:doc ->
    (s:{s:string | DocHost d s} * {dr:doc | DocHost dr s})

private extern API val getElementById: doc -> string -> elt

val getEltByTrackedIdT:
  p:({p:provs | Singleton p}) ->
  ({e:string | ExtensionId e}) ->
  d:({d:doc | exists (h:string) . (DocHost d h) && (InProvs (P h e) p)
  && (forall (s:string) . CanReadDOMId h s)}) ->
  tracked<string,p> ->
  tracked<elt,p>
let getEltByTrackedIdT p e d s =
  match s with | Tag us p -> (Tag (getElementById d us) p)

val getEltByIdT:p:
  ({p:provs | Singleton p}) ->
  ({e:string | ExtensionId e}) ->
  d:({d:doc | exists (h:string) . (DocHost d h) && InProvs (P h e) p}) ->
  ({s:string | exists (h:string) . (DocHost d h) && CanReadDOMId h s}) ->
  tracked<elt,p>
let getEltByIdT p e d s = Tag (getElementById d s) p

private extern API val getElementsByClass: doc -> string -> list<elt>

val getEltsByClassNameT:
  p:({p:provs | Singleton p}) ->
  ({e:string | ExtensionId e}) ->
```

```
  d:({d:doc | exists (h:string) . DocHost d h && InProvs (P h e) p}) ->
  ({s:string | exists (h:string) . DocHost d h && CanReadDOMClass h s}) ->
  list<tracked<elt,p>>
let getEltsByClassNameT p e d s = map (getElementsByClass d s) (fun x -> Tag x p)

extern API val installExternalInterface: ('a -> string -> unit) -> bool

extern API val getChildren:
  elt ->
  list<elt>

val getChildrenT:
  p:provs ->
  tracked<elt,p> ->
  tracked<list<elt>,p>
let getChildrenT p e = match e with | Tag ue p ->
  Tag (getChildren ue) p

extern API val getElValue: elt -> string

extern API val setValue
  :  elt
  -> s:string
  -> ({ ce2:elt | EltTextValue ce s })

val setValueT:
    e:elt ->
    ({p:provs | exists (s:string) . EltHost e s && AllCanCommunicateDOM p s}) ->
    tracked<string,p> ->
    mut_capability ->
    (unit * mut_capability)
let setValueT e p t c =
    let ce = Assume<elt, CanWriteValue> e in
    let x = match t with | Tag s p -> setValue ce s in
    ((), c)

private extern API val setAttribute: elt -> string -> string -> bool

extern API val getAttribute: elt -> string -> string

val setAttrT:
  ({elprov:provs | Singleton elprov}) ->
  ({h:string | exists (e:string) . InProvs (P h e) elprov}) ->
  ({p:provs | AllCanCommunicateDOM p h}) ->
  tracked<elt,elprov> ->
  k:string ->
  tracked<string,p> ->
  mut_capability ->
  (unit * mut_capability)
let setAttrT elprov p h el k v cap =
  match el with | Tag uel elprov ->
  match v with | Tag uv p ->
  let x = setAttribute uel k uv in
  ((), cap)

val getAttrT:
  p:provs ->
  tracked<elt,p> ->
  k:string ->
  tracked<string,p>
let getAttrT p te atname = match te with
  | Tag el p -> Tag (getAttribute el atname) p

val SetTextContent:
  ({elprov:provs | Singleton elprov}) ->
  ({h:string | exists (e:string) . InProvs (P h e) elprov}) ->
  ({p:provs | AllCanCommunicateDOM p h}) ->
  tracked<elt,elprov> ->
  tracked<string,p> ->
  mut_capability ->
  (unit * mut_capability)
let SetTextContent elprov h p el v cap =
  match el with | Tag uel elprov ->
  match v with | Tag uv p ->
  ((setTextContent uel uv), cap)

private extern API val makeXDocRequest: string -> string -> xdoc

val MakeXDocRequestT:
  h:({h:string | CanCommunicateXHR h}) ->
  t:string ->
```

```
242     eprin:string ->
243     ({p:provs | InProvs (P h eprin) p}) ->
244     mut_capability ->
245     (tracked<xdoc,p> * mut_capability)
246   let MakeXDocRequestT h t eprin pr m =
247     let r = (makeXDocRequest h t) in ((Tag r pr), m)
248
249   val MakeTrackedXDocRequestT: sp:provs ->
250     h:({h:string | AllCanCommunicateXHRTracked h sp}) ->
251     t:tracked<string,sp> ->
252     eprin:string ->
253     fp:({p:provs | forall (pr:prov) . (InProvs pr p) <=>
254       (InProvs pr sp || pr = (P h eprin))}) ->
255     mut_capability ->
256     (tracked<xdoc,fp> * mut_capability)
257   let MakeTrackedXDocRequestT sp h t eprin fp c =
258     match t with | Tag r sp -> (Tag (makeXDocRequest h r) fp, c)
259
260   private extern API val addNewDocHandler: string -> (doc -> unit) -> bool
261   val AddNewDocHandler:
262     ({s:string | CanHandleSites s}) ->
263     (doc -> unit) ->
264     unit
265   let AddNewDocHandler s f = let x = addNewDocHandler s f in ()
266
267   private extern API val addEventListener:
268     elt ->
269     string ->
270     (doc -> evt -> unit) ->
271     evtHandler
272
273   val addStatelessListener:
274     p:provs ->
275     tracked<elt,p> ->
276     ({s:string | AllCanCapture s p}) ->
277     (evt -> unit) ->
278     evtHandler
279   let addStatelessListener p e s f = match e with
280     | Tag ue p -> addEventListener ue s (fun d -> f)
281
282   val addListenerWithCap:
283     p:provs ->
284     tracked<elt,p> ->
285     ({s:string | AllCanCapture s p}) ->
286     (evt -> mut_capability -> unit) ->
287     mut_capability ->
288     (evtHandler * mut_capability)
289   let addListenerWithCap p t s f c = match t with
290     | Tag ue p -> ((addEventListener ue s (fun d e -> f e (let_mutate ()))), c)
291
292   private extern API val getEvTarget: e:evt -> ({d:doc | HasTarget e d})
293
294   val GetEvTarget:
295     p:provs ->
296     e:string ->
297     ({v:evt | exists (d:doc), (h:string) . HasTarget v d && DocHost d h && InProvs (P h e) p}) ->
298     tracked<doc,p>
299   let GetEvTarget p e d = Tag (getEvTarget d) p
300
301   extern API val ReadXDocEls: xdoc -> string -> (xelem -> bool) -> string -> list<string>
302
303   extern API val ClassifyText: string -> list<string>
304
305   extern API val StringConcat: string -> string -> string
306
307   extern API val ReadFileLines: ({s:string | CanReadLocalFile s}) ->
308                                 list<string>
309
310   extern API val Split: string -> string -> int -> string
311
312   extern API val StringToInt: string -> int
313
314   extern API val SetTimeout: int -> (unit -> unit) -> bool
315
316   extern API val ExtensionReturn:
317     dest:string ->
318     p:({p:provs | AllCanServeInformation dest p}) ->
319     tracked<'a,p> ->
320     bool
321
322   //
```

```
// Personal store
//

private extern API val addEntry:
  string ->
  string ->
  list<string> ->
  bool

val AddEntry:
  ({p:provs | AllCanUpdateStore p}) ->
  tracked<string,p> ->
  string ->
  tracked<list<string>,p> ->
  mut_capability ->
  (unit * mut_capability)
let AddEntry p e t cats c =
  match e with | Tag ue p ->
  match cats with | Tag ucats p ->
  let x = addEntry ue t ucats in
  ((), c)

private extern API val getStoreEntriesByTopic:
  string ->
  list<string>

val GetStoreEntriesByTopic:
  ({p:provs | AllCanReadStore p}) ->
  string ->
  tracked<list<string>,p>
let GetStoreEntriesByTopic p t = Tag (getStoreEntriesByTopic t) p

extern API val getTopInterests:
  ({p:provs | Singleton p && (exists (pp:prov) . pp = (P "repriv" "repriv") && InProvs pp p)}) ->
  int ->
  tracked<list<string>,p>
```