An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications

Dongseok Jang Ranjit Jhala Sorin Lerner Hovav Shacham

Dept. of Computer Science and Engineering
University of California, San Diego, USA
{d1jang,jhala,lerner,hovav}@cs.ucsd.edu

ABSTRACT

The dynamic nature of JavaScript web applications has given rise to the possibility of privacy violating information flows. We present an empirical study of the prevalence of such flows on a large number of popular websites. We have (1) designed an expressive, fine-grained information flow policy language that allows us to specify and detect different kinds of privacy-violating flows in JavaScript code, (2) implemented a new rewriting-based JavaScript information flow engine within the Chrome browser, and (3) used the enhanced browser to conduct a large-scale empirical study over the Alexa global top 50,000 websites of four privacy-violating flows: cookie stealing, location hijacking, history sniffing, and behavior tracking. Our survey shows that several popular sites, *including Alexa global top-100 sites*, use privacy-violating flows to exfiltrate information about users' browsing behavior. Our findings show that steps must be taken to mitigate the privacy threat from covert flows in browsers.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Experimentation, Languages

Keywords

privacy, web security, information flow, JavaScript, web application, dynamic analysis, rewriting, history sniffing

1. INTRODUCTION

JavaScript has enabled the deployment of rich browser-based applications that are fashioned from code sourced from different mutually distrusting and potentially malicious websites. However, JavaScript lacks language-based protection and isolation mechanisms and sports several extremely dynamic language features. Browser-level isolation policies like the same-origin policy for domain object model (DOM) objects are coarse-grained and do not uniformly apply to application resources. Consequently, the proliferation of JavaScript has also opened up the possibility of a variety of security vulnerabilities.

In this paper, we present an empirical study of the prevalence of an important class of vulnerabilities that we call *privacy-violating information flows.* This general class includes several different kinds of attacks that have been independently proposed and studied in the literature.

1. *Cookie Stealing:* Code included from a particular site, say for displaying an advertisement, has access to all the information on the hosting web page, including the cookie, the location bar, and any other sensitive information stored on the page. Thus, if the ad code is malicious it can cause the cookie and other sensitive pieces of information to be leaked to the third-party ad agencies, and can lead to a variety of routinely observed attacks like request forgery.

2. *Location Hijacking:* In a manner similar to the above case, the dynamically loaded untrusted code can influence the document's location, by influencing the values stored in URL string variables that are read to dynamically generate HTTP requests. Consequently, the dynamically loaded code can navigate the page to a malicious site that exploit a bug in the browser to fully compromise the machine [24] or mounts a phishing attack.

3. *History Sniffing:* In most browsers, all application domains share access to a single visited-page history, file cache, and DNS cache [12]. This leads to the possibility of history sniffing attacks [14], where a malicious site (say, `attacker.com`) can learn whether a user has visited a specific URL (say, `bankofamerica.com`), merely by inducing the user to visit `attacker.com`. To this end, the attack uses the fact that browsers display links differently depending on whether or not their target has been visited [6]. In JavaScript, the attacker creates a link to the target URL in a hidden part of the page, and then uses the browser's DOM interface to inspect how the link is displayed. If the link is displayed as a visited link, the target URL is in the user's history. `Tealium` and `Beencounter` sell services that allow a website to collect the browsing history of their visitors using history sniffing.

4. *Behavior Tracking:* The dynamic nature of JavaScript allows a website to construct a high-fidelity timeline of how a particular user interacted with a web page including, for

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

example, precise information about the user's mouse clicks and movements, scrolling behavior, and what parts of the text were highlighted, simply by including JavaScript event handlers that track mouse and keyboard activity. This information can then be sent back to the server to compute statistics about how users interact with a given web page. Several web-analytics companies sell products that exploit these flows to track information about users. For example, `ClickTale` allows websites to precisely track their users' mouse movements and compute aggregate heat maps based on where users move their mouse, and `tynt` allows websites to track what text is being copied from them. services allow websites to gather fined-grained information about the behaviors of their users without any indication to users that additional information gathering is taking place. We believe that users understand that page navigation (as a result of clicking on a link) causes information to be sent to the server, but do not believe they understand that other actions, like mousing over an image, can silently do the same. Verifying this belief will require a user study. We hope that the data we have collected will help inform the broader discussion about the privacy implications of such flows.

Privacy-violating information flows are not merely theoretical possibilities of academic interest. Indeed, the possibility of history sniffing has prompted a discussion spanning 8 years and over 261 comments in Bugzilla about preventing history sniffing in Firefox [3], which has finally culminated in a recent fix [2]. This lengthy process illustrates the importance of privacy-violating flows for Web 2.0 users and the difficulty of designing defenses against them without breaking legitimate websites.

Despite the knowledge that privacy-violating flows are possible and even likely, little is actually known about their occurrence in the wild. For example, how many websites extract this kind of information from their users? Are there any popular sites that do this? Do websites use pre-packaged solutions like `Tealium`, `Beencounter` and `ClickTale`? Or do they construct their own implementations? Are these implementations obfuscated to evade detection, and, if so, how? The lack of empirical data about the prevalence of privacy-violating flows has hampered the development and deployment of effective defenses against this increasingly important class of attacks. The main contribution of this paper is to provide concrete data to answer such questions through an exhaustive empirical evaluation of several privacy-violating flows in a large number of popular websites. We have carried out this study in three steps.

First, we have designed an expressive, fine-grained information flow policy language that allows us to specify and detect different kinds of privacy-violating flows in JavaScript code (Section 2.1). In essence, our language allows us to describe different privacy-violating flows by specifying sites within the code where taints are *injected* and sites from which certain taints must be *blocked*. For example, to specify a cookie stealing flow, we inject a "secret" taint into the cookie, and block that taint from flowing into variables controlled by third-party code. To specify a location hijacking flow, we inject an "untrusted" taint onto any values originating third-party code, and block that taint from flowing into the document's location field. To specify a history sniffing flow, we inject a "history" taint on the fields containing the style attributes of links, and block that taint from flowing into the parameters of methods that send messages over the network. To specify a behavior tracking flow, we inject "behavior" taint to the inputs of the handlers registered for events triggered by user behavior.

Second, we have implemented a new JavaScript information flow engine in the Chrome browser. Unlike previous JavaScript information flow infrastructures [10, 30, 8], our engine uses a dynamic source-to-source rewriting approach where taints are injected, propagated and blocked within the rewritten code (Section 2.2). Although the rewriting is performed inside the browser, implementing our approach requires understanding only the browser's AST data structure, and none of the complexity of the JavaScript runtime. Thus, in addition to supporting an extremely flexible flow policy specification language, our approach is simple to implement and can readily be incorporated inside other browsers. Even though the taints are propagated in JavaScript, as opposed to natively, the overhead of our approach is not prohibitively high. Our approach adds on average 60 to 70% to the total page loading time over a fast network (which is the worst condition to test our JavaScript overhead). This is efficient enough for our exploratory study, and with additional simple optimizations could even be feasible for interactive use (Section 3).

Third, we have used our modified version of the Chrome browser to conduct a large-scale empirical study over the Alexa global top 50,000 websites of four privacy-violating flows: cookie stealing, location hijacking, history sniffing and behavior tracking. Our results reveal interesting facts about the prevalence of these flows in the wild. We did not find any instances of location hijacking on popular sites, but we did find that there are several third party ad agencies to whom cookies are leaked. We found that several popular sites — *including an Alexa global top-100 site* — make use of history sniffing to exfiltrate information about users' browsing history, and, in some cases, do so in an obfuscated manner to avoid easy detection. We also found that popular sites, such as Microsoft's, track users' clicks and mouse movements, and that `huffingtonpost.com` has the infrastructure to track such movements, even though we did *not* observe the actual flow in our experiments. Finally, we found that many sites exhibiting privacy-violating flows have built their own infrastructure, and do not use pre-packaged solutions like `ClickTale`, `tynt`, `Tealium`, or `Beencounter`. Thus, our study shows that popular Web 2.0 applications like mashups, aggregators, and sophisticated ad targeting are rife with different kinds of privacy-violating flows, and, hence, there is a pressing need to devise flexible, precise and efficient defenses against them.

2. INFORMATION FLOW POLICIES

We present our approach for dynamically enforcing information flow policies through an example that illustrates the mechanisms used to generate, propagate and check taint information for enforcing flow policies. The focus of our information flow policies and enforcement mechanism is to detect many privacy-violating flows in the wild, not to provide a bullet-proof protection mechanism (although our current system could eventually lead to a protection mechanism, as discussed further in Section 7). For space reasons, we defer a formal treatment of our rewriting algorithm to a technical report [15].

```
var initSettings = function(s){
  searchUrl = s;
}

initSettings("a.com");

var doSearch = function() {
   var searchBox = getSearchBoxValue();
   var searchQry = searchUrl + searchBox;
   document.location = searchQry;
}

eval(load("http://adserver.com/display.js"));
```

Figure 1: JavaScript code from a website `a.com`.

Web page Consider the JavaScript in Figure 1. Suppose that this code is a distillation of the JavaScript on a web page belonging to the domain `a.com`. The web page has a text box whose contents can be retrieved using a call to the function `getSearchBoxValue` (not shown). The function `initSettings` is intended to be called once to initialize settings used by the page. The `doSearch` function is called when the user clicks a particular button on the page.

Dynamically Loaded Code The very last line of the code in Figure 1 is a call to `load`() which is used to dynamically obtain a string from `adserver.com`. This string is then passed to `eval`() which has the effect of "executing" the string in order to update the web page with an advertisement tailored to the particular user.

Malicious Code Suppose that the string returned by the call to `adserver.com` was:

```
initSettings("evil.com");
```

When this string is passed to `eval`() and executed, it overwrites the page's settings. In particular, it sets the variable `searchUrl` which is used as the prefix of the query string, to refer to an attacker site `evil.com`. Now, if the user clicks the search button, the `document.location` gets set to the attacker's site, and thus the user is redirected to a website which can then compromise her machine. Similarly, dynamically loaded code can cause the user to leak their password or other sensitive information.

2.1 Policy Language

The flexibility and dynamic nature of JavaScript makes it difficult to use existing language-based isolation mechanisms. First, JavaScript does not have any information hiding mechanisms like private fields that could be used to isolate `document.location` from dynamically loaded code. Indeed, a primary reason for the popularity of the language is that the absence of such mechanisms makes it easy to rapidly glue together different libraries distributed across the web. Second, the asynchronous nature of web applications makes it difficult to enforce isolation via dynamic stack-based access control. Indeed, in the example above, the malicious code has done its mischief and departed well before the user clicks the button and causes the page to relocate.

Thus, to reconcile safety and flexible, dynamic code composition, we need fine-grained isolation mechanisms that prevent untrusted code from viewing or affecting sensitive data. Our approach to isolation is information flow control [11, 21], where the isolation is ensured via two steps. First, the website's developer provides a fine-grained *policy* that describes which values *can affect* and *be affected* by others. Second, the language's compiler or run-time *enforce* the policy, thereby providing fine-grained isolation.

Policies In our framework a fine-grained information flow policy is specified by defining taints, injection sites and checking sites. A *taint* is any JavaScript object, *e.g.,* a URL string denoting the provenance of a given piece of information. A *site*

$$r.f(x\ldots)$$

corresponds to the invocation of the method f with the arguments $x\ldots$, on the receiver object r. Such site expressions can contain concrete JavaScript (*e.g.,* `document.location`), or pattern variables (*e.g.,* \$1) that can match against different concrete JavaScript values, and which can later be referenced.

In order to allow sites to match field reads and writes, we model these using getter and setter methods. In particular, we model a field read using a call to method `getf`, which takes the name of the field as an argument, and we model a field write using a call to a method `setf`, which takes the name of a field and the new value as arguments. To make writing policies easier, we allow some simple syntactic sugar in expressing sites: $r.x$ used as an r-value translates to $r.\texttt{getf}(x)$ and $r.x = e$ translates to $r.\texttt{setf}(x, e)$.

An *injection* site

$$\textsf{at } S \textsf{ if } P \textsf{ inject } T$$

stipulates that the taint T be *added to* the taints of the object output by the method call described by the site S as long as the condition P holds at the callsite. For example, the following injection site unconditionally injects a "secret" taint at any point where `document.cookie` is read:

$$\textsf{at document.cookie if } \texttt{true} \textsf{ inject "secret"}$$

To make the policies more readable, we use the following syntactic sugar: when "if P" is omitted, we assume "if `true`". As a result, the above injection site can be expressed as:

$$\textsf{at document.cookie inject "secret"} \qquad (1)$$

A *checking* site

$$\textsf{at } S \textsf{ if } P \textsf{ block } T \textsf{ on } V$$

stipulates that at site S, if condition P holds, the expression V must *not contain* the taint T. We allow the guard (P), the taint (T) and the checked expression (V) to refer to the any pattern variables that get bound within the site S. As before, when "if P" is omitted, we assume "if `true`".

For example, consider the following checking site:

$$\textsf{at } \$1.x{=}\$2 \textsf{ if } \$1.\texttt{url} \neq \textsf{"a.com" block "secret" on } \$2 \quad (2)$$

The `url` field referenced above is a special field added by our framework to every object, indicating the URL of the script that created the object. The above checking site therefore ensures that no value tainted with "secret" is ever assigned into an object created by a script that does not originate from `a.com`.

Confidentiality policies can be specified by injecting a special "secret" taint to the getter-methods for confidential variables, and checking that such taints do not flow into the inputs of the setter-methods for objects controlled by code originating at domains other than a.com. Such a policy could be formally specified using (1) and (2) above.

Integrity policies can be specified by injecting special "untrusted" taints to getter-methods for untrusted variables, and checking that such taints do not flow into the inputs of setter-methods for trusted variables. Such a policy could be formally specifed as:

at $\$1.x$ if $\$1.$url $\neq$ "a.com" inject "untrusted"

at document.location $= \$1$ block "untrusted" on $\$1$

We can specify even finer grained policies by refining the taints with information about individual URLs. The expressiveness of our policy language allows us to quickly experiment with different kinds of flows within the same basic framework, and could also lay the foundation for a browser-based protection mechanism.

2.2 Policy Enforcement

The nature of JavaScript and dynamic code loading makes precise static policy enforcement problematic, as it is impossible to predict what code will be loaded at run-time. Thus, our approach is to carry out the enforcement in a fully dynamic manner, by rewriting the code in order to inject, propagate and checks taints appropriately.

Although there are known dangers to using rewriting-based approaches for protection [20], our current goal is actually not protection, bur rather to find as many privacy-violating flows as possible. As such, one of our primary concerns is ease of prototyping and flexibility; in this setting, rewrite-based approaches are very useful. In particular, implementing our approach only required understanding the browser's AST data structure, and none of the complexities of the JavaScript runtime, which allowed us to quickly build and modify our prototype as needed. Furthermore, keeping our framework clearly separate from the rest of the browser gives us the flexibility of quickly porting our approach to new versions of Chrome, or even different browsers.

Policy Enforcement Our framework automatically rewrites the code using the specified injection and checking sites to ensure that taints are properly inserted, propagated and checked in order to enforce the flow policy. First, we use the checking (resp. injection) sites to synthesize wrappers around the corresponding methods that ensure that the inputs only contain (resp. outputs are tainted with) the taints specified by the corresponding taint expressions whenever the corresponding guard condition is met. Second, we rewrite the code so that it (dynamically) propagates the taints with the objects as they flow through the program via assignments, procedure calls *etc.*. We take special care to ensure correct propagation in the presence of tricky JavaScript features like eval, prototypes, and asynchronous calls.

Rewriting Strategy Our strategy for enforcing flow policies, is to extend the browser with a function that takes a code string and the URL from which the string was loaded, and returns a rewritten string which contains operations that perform the injection, propagation and checking of taints. Thus, to enforce the policy, we ensure that the code on the web page is appropriately rewritten before it is evaluated. We ensure that "nested" eval-sites are properly handled as follows. We implement our rewriting function as a procedure in the browser source language (*e.g.,* C++) that can be called from within JavaScript using the name RW and the rewriter wraps the arguments of eval within a call to RW to ensure they are (recursively) rewritten before evaluation [32].

```
//var initSettings = function(){...}
tmp0 = box(function(s){searchUrl = s;}, "a.com"),
var initSettings = tmp0;

//initSettings("a.com");
tmp1 = box("a.com", "a.com"),
initSettings(tmp1);

//var doSearch = function(){...}
var doSearch = box(function(){
  var searchBox = getSearchBoxValue();

  //var searchQry = searchBox + searchUrl;
  var searchQry = TSET.direct.add(searchUrl),
                  tmp2 = unbox(searchUrl),

                  TSET.direct.add(searchBox),
                  tmp3 = unbox(searchBox),

                  tmp4 = tmp2 + tmp3,
                  TSET.boxAndTaint(tmp4, "a.com");

  //document.location = searchQry;
  check(searchQry, "untrusted"),
  document.location = searchQry;
  }, "a.com");

//eval(load("http://adserver.com/display.js"));
tmp5 = box("http://adserver.com/display.js", "a.com"),
tmp6 = box(load(tmp5), "www.a.com"),
tmp6.url = tmp5,
eval(RW(tmp6, tmp6.url));
```

Figure 2: Rewritten code from a.com. The comments above each block denote the original version of the rewritten block.

When the rewriting procedure is invoked on the code from Figure 1 and the URL a.com, it emits the code shown in Figure 2. The rewriting procedure rewrites each statement and expression. (In Figure 2, we write the original code as a comment above the rewritten version.) Next, we step through the rewritten code to illustrate how taints are injected, checked and propagated, for the integrity property that specifies that document.location should only be influenced by a.com.

Injection To inject taints, we extend *every* object with two special fields url and taint. To achieve this, we wrap all object creations inside a call to a special function box which takes a value and a *url* and creates a boxed version of the value where the url field is set to *url* indicating that the object originated at *url*, and the taint field is set to the empty set of taints. We do this uniformly for all objects, including functions (*e.g.,* the one assigned to initSettings), literals (*e.g.,* the one passed as a parameter to initSettings), *etc.*. Next, we use the specified injection sites to rewrite the code in order to (conditionally) populate the taint fields at method calls that match the sites. However, the integrity injection site does not match anywhere in the code so far, and so no taints are injected yet – they will be injected when code gets loaded from the ad server.

Checking Before each call site that matches a specified check site, we insert a call to a special `check` function. The call to `check` is predicated on the check site's condition. The function `check` is passed the checked expression V and taint T corresponding to the matching check site. The function determines whether the taints on the checked expression contain the prohibited taint, and if so, halts execution.

For example, consider the rewritten version of the assignment to `document.location` in the body of `doSearch` which matches the checking site from the integrity policy. The rewrite inserts a call to `check`. At run-time, when this call is executed it halts the program with a flow-violation message if `searchQry.taint` has a (taint) value of the form "`untrusted`".

Propagation Next, we consider how the rewriting instruments the code to add instructions that propagate the taints.

- For assignments and function calls, as all objects are boxed, the taints are carried over directly, once we have created temporaries that hold boxed versions of values. For example, the call to `initSettings` uses `tmp0`, the boxed version of the argument, and hence passes the taints into the function's formals. The assignment to `searchBox` is unchanged from before, as the right-hand side is function call (whose result has already been appropriately boxed).
- For binary operations, we must do a little more work, as many binary operations (*e.g.,* string concatenation) require their arguments be *unboxed.* To handle such operations, we extend the code with a new object called the *taint-set*, named TSET. We use this object to accumulate the taints of sub-expressions of compound expressions. The object supports two operations. First, TSET.`direct.add`(x, url), which adds the taints in `x.taint` to the taint-set. Second, TSET.`boxAndTaint`(x, url), which creates a boxed version of `x` (if it is not boxed), and the taints accumulated on the taint-set, clears the taint-set, and returns the boxed-and-tainted version of `x`. We use the `direct` field as there are several other uses for the TSET object that are explained later. For example, consider the rewritten version of `searchBox + searchUrl`. We add the taints from `searchBox` (resp. `searchUrl`) to the taint-set, and assign an unboxed version to the fresh temporary `tmp2` (resp. `tmp3`). Next, we concatenate the unboxed strings, and assign the result to `tmp4`. Finally, we call TSET.`boxAndTaint`(`tmp4`, "`a.com`"), which boxes `tmp4`, adds the taints for the sub-expressions stored in the taint-set and returns the boxed-and-tainted result.
- For code loading operations (modeled as `load`(·)), the rewriting boxes the strings, and then adds a `url` field to the result that indicates the domain from which the string was loaded. For example, consider the code loaded from `adserver.com`. The name of the URL is stored in the temporary `tmp5`, and the boxed result is stored in a fresh temporary `tmp6`, to which we add a `url` field that holds the value of `tmp5`.
- For `eval` operations, our rewriting interposes code that passes the string argument to `eval` and the URL from which the string originated to the the rewriting function RW, thereby ensuring the code is rewritten before it is evaluated. For example, consider the operation at the last line of the code from Figure 1 which `eval`'s the string loaded from `adserver.com`. In the rewritten version, we have a boxed version of the string stored in `tmp6`; the rewriting ensures that the string that gets executed is actually `tmp6` rewritten assuming it originated at `tmp6.url`, which will have the effect of ensuring that taints are properly injected, propagated and checked within the dynamically loaded code.

The above code assumes, for ease of exposition, that the fields `taint` and `url` are not read, written or removed by any code other than was placed for tracking.

Attack Prevention Suppose that the `load`(·) operation returns the string `initSettings("evil.com")`. The rewritten code invokes the rewriting function RW on the string, and the URL `adserver.com` yielding the string

```
tmp10 = box("evil.com", "adserver.com"),
if (tmp10.url != "a.com"){
  tmp10.taint += ["untrusted"]
},
initSettings(tmp10);
```

The `if`-expression injects the taints to the value returned by the implicit getter-call (*i.e.,* the read of `tmp10`) that yields the value passed to `initSettings`. Thus, the argument passed to `initSettings` carries the taint "`untrusted`", which flows into `searchUrl` when the assignment inside `initSettings` is executed. Finally, when the button click triggers a call to `doSearch`, the taint flows through the taint-set into the value returned by the call TSET.`boxAndTaint`(`tmp4`, "`a.com`"), and from there into `searchQry`. Finally, the `check` (just before the assignment to `document.location`) halts execution as the flow violates the integrity policy, thereby preventing the redirection attack.

Rewriting for Confidentiality Policies The above example illustrates how rewriting enforces integrity policies. The case for confidentiality policies differs only in how taints are injected and checked; the taints are propagated in an identical fashion. To *inject* taints, the rewriting adds a "`secret`" taint to the results of each *read* from a confidential object (*e.g.,* `document.cookie`.) To *check* taints, the rewriting inserts calls to `check` before any writes to variables (*i.e.,* invocations of setter methods) in code originating in untrusted URLs. The `check` halts execution before any value with the "`secret`" taint can flow into an untrusted location.

Robustness Even though the primary purpose of our tool so far has been to evaluate existing flows (a scenario under which we don't need to worry about malicious code trying to subvert our system), our framework does in fact protect its internal data structures from being maliciously modified. In particular, our tool disallows a user JavaScript program from referencing any of the variables and fields used for taint tracking such as TSET. More specifically, since our framework tracks reads and writes to *all* variable and field, it can simply stop a program that tries to read or write to any of the internal variables that we use to track information flow.

2.3 Indirect Flows

Next, we look at how the rewriting handles indirect flows due to control dependencies. We start with the data structure that dynamically tracks indirect flows, and then describe the key expressions that are affected by indirect flows.

Indirect Taint Stack (TSET.indirect) To track indirect flows, we augment the taint set object with an *indirect-taint* stack (named TSET.indirect). Our rewriting ensures that indirect taints are added and removed from the indirect taint stack as the code enters and leaves blocks with new control dependences. The TSET.boxAndTaint(·, ·) function, which is used to gather the taints for the RHS of assignments, embellishes the (RHS) object with the direct taints at the *top of* the direct taint stack, *and* the indirect taints stored *throughout* the indirect taint stack. The latter ensures that at each assignment also propagates the indirect taints that held at the point of the assignment.

Branches For branch expressions of the form if e_1 e_2 e_2, we first assign the rewritten guard to a new temporary tmp$_1$, and push the taints on the guard onto the indirect taint stack. These taints will reside on the indirect taint stack when (either) branch is evaluated, thereby tainting the assignments that happen inside the branches. After the entire branch expression has been evaluated, the rewritten code pops the taints, thereby reverting the stack to the set of indirect taints before the branch was evaluated.

Example Consider the branch expression: if (z) { x = 0 } To ensure that taints from z flow into x when the assignment occurs inside the then-branch, the expression is rewritten to:

```
tmp = z,
TSET.indirect.push(tmp),
if (unbox(tmp)){
  x = TSET.boxAndTaint(box(0,...),...)
},
TSET.indirect.pop()
```

The ellipses denote the URL string passed to RW and we omit the calls to check and TSET.direct.add(·, ·) for brevity. The rewrite ensures that the taints from the guard z are on the indirect taint stack inside the branch, and these taints are added to the (boxed version of) 0 that is used for the assignment, thereby flowing them into x. The pop after the branch finishes reverts the indirect stack to the state prior to the branch.

Indirect vs. Implicit Flows. The above example illustrates a limitation of our fully dynamic approach; we can track *indirect* flows induced by a taken branch (such as the one above) but not *implicit* flows that occur due to a not-taken branch. For example, if the above branch was preceded by an assignment that initialized x with 1, then an observer that saw that x had the value 1 after the branch would be able to glean a bit of information about the value of z. Our rewriting, and indeed, any fully dynamic analysis [7] will fail to detect and prohibit such implicit flows.

Function Calls Our rewriting adds an indirect taint parameter to each function definition. This parameter holds the indirect taints that hold at the start of the function body; the taints on it are pushed onto the indirect taint stack when the function begins execution. Furthermore, the rewriting ensures that at each function callsite, the indirect taints (in the indirect taint stack) at the caller are passed into the indirect taint parameter of the callee.

Event Handlers cause subtle information flows. For example, if foo is the handler for the MouseOver event, the fact that foo is executed contains the information that the mouse hovered over some part of the page. We capture these flows as indirect flows triggered by the tests within the DOM event dispatch loop

```
while(1){
  e = getEvent();
  if (e.isClick()) onClick(e);
  if (e.isMouseOver()) onMouseOver(e);
  if (e.isScroll()) onScroll(e);
  ...
}
```

Thus, to capture flows triggered by a MouseOver event, we simply inject a taint at the output of the $1.isMouseOver(...). The if ensures that the taint flows into the indirect taint parameter of the registered handler (bound to onMouseOver).

Full JavaScript The examples described in this section have given a high-level overview of our rewriting-based approach to enforcing information flow policies. Our implementation handles all of JavaScript including challenging language features like prototypes, with-scoping, and higher-order functions. Our implementation also incorporates several subtle details pertaining to how taints can be stored and propagated for *unboxed* objects, to which a taint field cannot be added. The naive strategy of boxing all objects breaks several websites as several DOM API functions require unboxed objects as arguments. We refer the reader to an accompanying technical report [15] for the details.

3. IMPLEMENTATION AND PERFORMANCE EVALUATION

This section presents our implementation of rewrite-based information flow framework for JavaScript in the Chrome browser, and describes several experiments that quantify the performance overhead of our approach.

Implementation We implement the rewriting function as a C++ method (within Chrome) that is invoked on any JavaScript code just before it gets sent into the V8 execution engine. Thus, our implementation rewrites *every* piece of JavaScript including that which is loaded by <script> tags, executed by eval or executed by changing the web page via document.write. The TSET library is implemented in pure JavaScript, and we modified the resource loader of Chrome to insert the TSET library code into every JavaScript program it accesses. The TSET library is inserted into each web page as ordinary JavaScript using a <script> tag before any other code is loaded. The flow-enhanced Chrome can run in normal mode or in taint tracking mode. When the taint tracking is on, the modified Chrome tracks the taint flow as a user surfs on websites.

Optimizations We describe the three most important optimizations we performed for the "optimized" bar. The first and most important optimization is that we implemented the two most frequently executed TSET methods using 65 lines of C++, namely the methods for taint lookup and unboxing. Second, in the TSET.direct stack, when there is a pop followed by a push, and just before the pop there are *no* taints stored at the top of the stack, we cache the object at the top of the stack before poping, and then reuse that same object at the next push, thus avoiding having to create a new object. Because the push is called on every assignment, removing the object creation provides a significant

Site and rank	Total KLOC	Other KLOC			# Taint Val(k)			Cookie			Location		
		s	d	w	s	d	w	s	d	w	s	d	w
1. google	1.8	0	-	-	0	-	-	×	×	×	×	×	×
2. yahoo	7.4	7.0	-	0	29.5	-	0	×	×	×	×	×	×
3. facebook	9.1	9.1	9.1	9.1	12.4	9.3	8.4	×	×	×	×	×	×
4. youtube	7.5	7.3	7.3	5.7	21.0	20.7	20.7	✓	✓	✓	×	×	×
5. myspace	12.2	11.9	-	8.6	35.3	-	28.4	✓	✓	✓	×	×	×
6. wikipedia	<0.1	0	-	-	0	-	-	×	×	×	×	×	×
7. bing	0.7	0	-	-	0	-	-	×	×	×	×	×	×
8. blogger	1.8	1.1	-	0	0.8	-	0.2	✓	✓	×	×	×	×
9. ebay	13.6	13.4	-	12.9	244.9	-	244.9	✓	✓	✓	×	×	×
10. craigslist	0	0	-	-	0	-	-	×	×	×	×	×	×
11. amazon	5.3	4.8	-	-	40.1	-	-	×	×	×	×	×	×
12. msn	7.3	6.7	6.1	5.6	462.4	462.3	462.3	✓	×	×	×	×	×
13. twitter	5.6	5.5	-	1.3	48.2	-	38.3	×	×	×	×	×	×
14. aol	12.7	9.6	-	<0.1	129.8	-	60.9	✓	✓	×	×	×	×
15. go	1.1	0.9	0.2	-	76.3	2.0	-	✓	×	×	×	×	×
–. Average	8.2	6.1	4.5	3.0	63.1	52.7	35.9	48	38	17	0	0	0

Figure 3: Flow results for a subset of the Alexa global 100 (last row summarizes results for all 100 sites).

savings. Third, we also cache field reads in our optimized `TSET` library. For example, whenever a property `a.b` is referenced several times in the library, we store the value of the property in a temporary variable and reuse the value again. This produces significant savings, despite the fact that all our measurements used the JIT compiler of the V8 engine.

Benchmarks We employ a set of stress experiments using the standard cookie confidentiality and location integrity policies to evaluate the efficiency of our approach and the effect of optimizations. These policies require a significant amount of taint propagation, as the cookie and location properties are heavily accessed. As our benchmarks, we use the front pages on the websites from the latest Alexa global top 100 list. Alexa is a company which ranks websites based on traffic. The websites on the Alexa global top 100 vary widely in size and how heavily they use JavaScript, from 0.1 KLOC to 31.6 KLOC of JavaScript code. We successfully ran our dynamic analysis on all of the pages of Alexa global top 100 list, and we visited many of them manually to make sure that they function properly.

3.1 Policies

To measure efficiency, we checked two important policies on each site. First, `document.cookie` should remain confidential to the site. Second, `document.location` should not be influenced by another site. Both policies depend on a definition of what "another site" is. Unfortunately, using exactly the same URL or domain name often leads to many false alarms as what looks like a single website is in fact the agglomeration of several different domain names. For example, `facebook.com` refers to `fbcdn.net` for many of their major resources, including JavaScript code. Moreover, there are relatively well known and safe websites for traffic statistics and advertisements, which are referenced on many other websites, and one may want to consider those as safe. Thus, we considered three URL policies (*i.e.,* three definitions for "another site") (1) the *same-origin policy* stating that any website whose hostname is different from the hostname of the current one is considered a different site. (2) the *same-domain policy*, which is the same as the same-origin policy, except that websites from the same domain are considered to be the same (*e.g.,* `ads.cnn.com` is considered the same as `www.cnn.com`). (3) the *white-list policy*, which is the same as the same-domain policy, except that there is a global list of common websites that are considered the same as the source website. For our experiments, we treat websites referenced by three or more different Alexa benchmarks as safe. The white-list mainly consists of statistics and advertisement websites. We use a whitelist only to evaluate the performance of our system under such conditions; we leave the exact criteria for trusting a given third-party site to future work. Our rewriting framework makes it trivial to consider different URL policies; we need only alter the notion of URL equality in the checks done inside `TSET.boxAndTaint` and `TSET.check`.

Detected Flows Figure 3 shows the results of running our dynamic information framework on the Alexa global top 100 list using the above policies. Because of space constraints, we only show a subset of the benchmarks, but the average row is for *all* 100 benchmarks.

The columns in the table are as follows: "Site and rank" is the name of the website and its rank in the Alexa global 100 list; "Total KLOC" is the number of lines of JavaScript code on each website, including code from other sites, as formatted by our pretty printer; "Other KLOC" is the number of lines of code from other sites; "# Taint Val" is the number of dynamically created taint values; "Cookie" describes the `document.cookie` confidentiality policy, and "Location" describes the `document.location` integrity policy: ✓indicates policy violation, and ×indicates no flow *i.e.,* policy satisfaction.

The above columns are sub-categorized into three subcolumns depending on the applied URL policy: "s" is for the same-origin policy; "d" is for the same-domain policy; "w" is for the white-list policy. A dash in a table entry means that the value for that table entry is the same as the entry immediately to its left.

The code for each website changes on each visit. Thus, we ran our enhanced Chrome 10 times on each website. To gain confidence in our tool, we manually inspected every program on which a flow is detected, and confirmed that every flow was indeed real.

Variation based on URL policies The number of lines of code from other sites decreases as we move from the same-origin policy to the same-domain policy to the white-list policy. Note that in some cases, for example `facebook`, code from other sites is almost the same as the total lines of code.

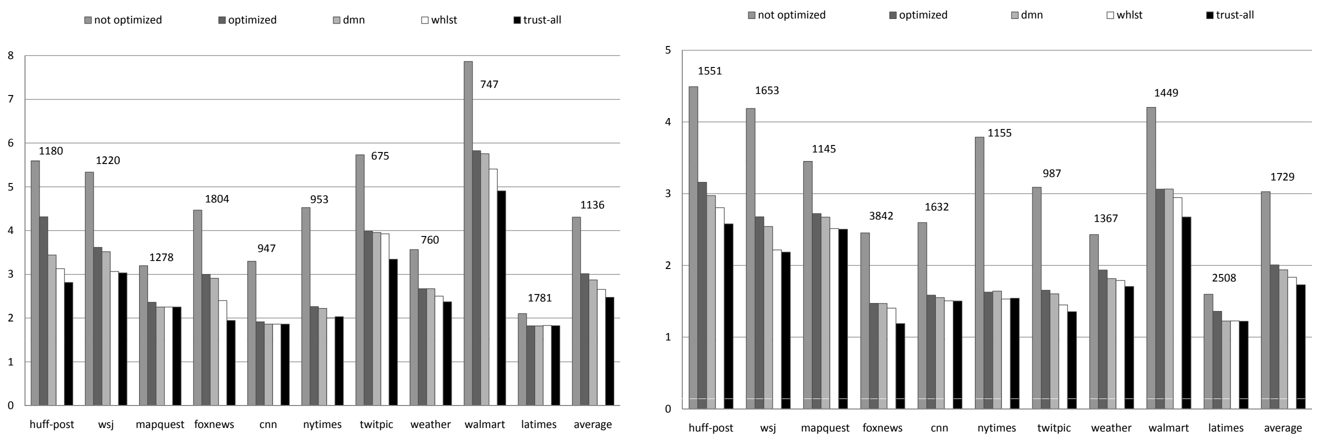

Figure 4: Slowdown for JavaScript (left) and Page Loading (right)

This is because most of the JavaScript code for `facebook` comes from a website `fbcdn.net`. This website is not in the same domain as `facebook`, and it is only referenced by one website and hence, not included in our whitelist. In such situations, a *site-specific* white-list would help, but we have not added such white-lists because it would be difficult for us to systematically decide for all 100 benchmarks what these white-lists should be. Thus, as we do not use site-specific white-lists, our policy violations may not correspond to undesirable flows.

As the amount of other-site code decreases as we move from "s" to "d" to "w", the number of dynamically created taint values also decreases, at about the same rate. That is, a large drop in other-site code leads to a correspondingly large drop in the number of taint values created. Moreover, as expected, the number of policy violations also decreases, as shown on the last line of the table: the violations of the `document.cookie` policies goes from 48 to 38 to 17. We did not see a violation of the `document.location` policy in any of our benchmarks.

3.2 Timing Measurements

Our rewrite-based information flow technique performs taint-tracking dynamically, and so it is important to evaluate the performance overhead of our approach. We measure performance using two metrics: total page load time, and JavaScript run time. We modified the Chrome browser to allow us to measure for each website (1) the time spent executing JavaScript on the site, and (2) the total time spent to download and display the site. Figures 4 describes our timing measurements for JavaScript time, and total download time on the 10 benchmarks with the largest JavaScript code bases. The measurements were performed while tracking both the `document.cookie` confidentiality and `document.location` integrity policies. The "average" benchmark represents the average time over all 10 benchmarks. For each benchmark there are five bars which represent running time, so smaller bars mean faster execution. For each benchmark, the 5 bars are normalized to the time for the unmodified Chrome browser for that benchmark. Above each benchmark we display the time in milliseconds for the unmodified Chrome browser (which indicates what "1" means for that benchmark). The left most bar "not-optimized" represents our technique using the original version of our `TSET` library, and using the same-origin URL policy. For the remaining bars, each bar represents a single change from the bar immediately to its left: "optimized" uses a hand-optimized version of our `TSET` library, rather than the original version; "dmn" changes the URL policy to same-domain; "whlist" changes the URL policy to white-list; and "trust-all" changes the URL policy to the trivial policy where *all* websites are trusted.

JavaScript execution time The left chart of Figure 4 shows just the JavaScript execution time. As expected, the bars get shorter from left-to-right; from "not-optimized" to "optimized", we are adding optimizations; and then the remaining bars consider progressively more inclusive URL policies meaning there are fewer taints to generate, propagate and check.

The data from Figure 4 shows that our original `TSET` library slows down JavaScript execution significantly – anywhere from about 2.1X to 7.9X, and on average about 4.3X. The optimized `TSET` library provides significant performance gains over the original library and provides 3.0X slowdown. The various white-lists provide some additional gain, but the gain is relatively small. To understand the limits of how much white-lists can help, we use the "trust-all" bar, which essentially corresponds to having a white-lists with every website on it. Overall, it seems that even in the best case scenario, white-lists do not help much in the overhead of our approach. This is because our approach needs to track the flow of `cookie` regardless of the number of external sites.

Total execution time The right chart of Figure 4 shows the *total* execution time of the enhanced Chrome while loading the web page and running the scripts on it. These measurements were collected on a fast network at a large university. The faster the network, the larger the overheads in Figure 4 will be, as the time to download the web page can essentially hide the overhead of running JavaScript. Thus, by using a fast network, Figure 4 essentially shows some of the worst case slowdowns of our approach. Here again, we see that the "optimized" bar is significantly faster than the "not-optimized" bar. We can also see that the "whlst" bar provides a loading experience that is about 73% slower.

4. EMPIRICAL STUDY OF HISTORY HIJACKING

Next, we present an empirical study of the prevalence of history hijacking on popular websites. Recall that links corresponding to URLs visited by the user are typically rendered differently than links corresponding to URLs not visited by the user. In essence, the attack works by inserting invisible links into the web page and having JavaScript inspect certain style properties of links, for example the color field, thereby determining whether the user has visited a particular URL. While researchers have known about the possibility of such attacks, hitherto it was not known how prevalent they are in real, popular websites. We have used our JavaScript information flow framework to detect and study the prevalence of such attacks on a large set of websites, and show that history hijacking is used, even by quite popular websites.

Policies We formalize history hijacking in our framework using the following information flow policies:

```
at $1.getCompStyle($2,...) if $2.isLink() inject "secret"
at document.send($1,$2) block "secret" on $2
```

In particular, whenever the computed style of a link is read using `getCompStyle`, the return value is marked as secret. Whenever a value is sent on the network using `document.send`, the second parameter (which is the actual data being sent) should not be tainted.

Benchmarks and Summary of Results To evaluate the prevalence of history hijacking, we ran our information flow framework using the above two policies on the front pages of the Alexa global top 50,000 websites.[1] To visit these sites automatically, we implemented a simple JavaScript web page that directs the browser to each of these websites. We successfully ran our framework on these sites in a total of about 50 hours. The slowdown for history sniffing was as follows: the JavaScript code slowed down by a factor 2.4X and total page loading time on a fast network increased by 67%. Overall, we found that of these 50,000 sites, 485 of them inspect style properties that can be used to infer the browser's history. Out of 485 sites, 63 are reported as transferring the browser's history to the network, and we confirmed that 46 of them are actually doing history hijacking, one of these sites being in the Alexa global top 100.

Real cases of history sniffing Out of 63 websites reported as transferring the browser's history by our framework, we confirmed that the 46 cases were real history sniffing occurrences. Table 1 lists these 46 websites. For each history-sniffing site, we give its Alexa rank, its URL, a description of the site, where the history-sniffing code comes from, and a list of some of the URLs inspected in the browser history.

Rank	Site	Desc	Src	Inspected URLs
61	`youporn`	adult	H	pornhub,tube8,+21
867	`charter.net`	news	I	cars,edmunds,+46
2333	`feedjit`	traffic	F	twitter,facebook,+6
2415	`gamestorrents`	fun	M	amazon,ebay,+220
2811	`newsmax`	news	I	cars,edmunds,+46
3508	`namepros`	forum	F	twitter,facebook,+6
3603	`fulltono`	music	M	amazon,ebay,+220
4266	`youporngay`	adult	H	pornhub,tube8,+21
4581	`osdir`	tech	I	cars,edmunds,+46
5233	`gamesfreak`	fun	I	cars,edmunds,+46
5357	`morningstar`	finance	I	cars,edmunds,+46
6500	`espnf1`	sports	I	cars,edmunds,+46
7198	`netdoctor`	health	I	cars,edmunds,+46
7323	`narutocentral`	fun	I	cars,edmunds,+46
8064	`subirimagenes`	hosting	M	amazon,ebay,+220
8644	`fucktube`	adult	H	tube8,xvideos,+9
9616	`straightdope`	news	I	cars,edmunds,+46
10152	`guardafilm`	movie	M	amazon,ebay,+220
10415	`estrenosdtl`	movie	M	amazon,ebay,+220
11330	`bgames`	fun	I	cars,edmunds,+46
12084	`10best`	travel	I	cars,edmunds,+46
12164	`twincities`	news	I	cars,edmunds,+46
16752	`kaushik.net`	blog	H	facebook,+100
17379	`todocvcd`	content	M	amazon,ebay,+220
17655	`filmannex`	movie	I	cars,edmunds,+46
17882	`planet-f1`	sports	I	cars,edmunds,+46
18361	`trailersplay`	movie	M	amazon,ebay,+220
20240	`minyanville`	finance	I	cars,edmunds,+46
20822	`pixmac`	hosting	H	istockphoto,+27
22010	`fotoflexer`	widget	I	amazon,ebay,+220
23577	`xepisodes`	fun	M	amazon,ebay,+220
23626	`s-p`*	movie	F	facebook,youtube,+8
24109	`mimp3.net`	music	M	amazon,ebay,+220
24414	`allaccess`	news	I	amazon,ebay,+220
24597	`petitchef`	food	M	amazon,ebay,+220
24815	`bleachcentral`	fun	I	amazon,ebay,+220
25750	`hoopsworld`	sports	I	amazon,ebay,+220
27366	`net-games.biz`	fun	I	cars,edmunds,+46
31638	`6speedonline`	car	I	cars,edmunds,+46
34661	`msgdiscovery`	tech	M	amazon,ebay,+220
35773	`moneynews`	finance	I	cars,edmunds,+46
37333	`a-g`*	religion	H	facebook,+62
41490	`divxatope`	content	M	amazon,ebay,+220
45264	`subtorrents`	content	M	amazon,ebay,+220
48284	`sesionvip`	movie	M	amazon,ebay,+220
49549	`youporncocks`	adult	H	pornhub,tube8,+21

Table 1: Websites that perform real sniffing. Top-level domains are `.com` if not otherwise specified. `s-p` and `a-g` abbreviate `sincortespublicitarios.com` and `answersingenesis.org`. "Src" is the source of the history sniffing JavaScript code: "I", "M" and "F", indicate the code came from `interclick.com`, `meaningtool.com`, and `feedjit.com` respectively, and "H" indicates the code came from the site itself.

Each one of the websites in Table 1 extracts the visitor's browsing history and transfers it to the network. Many of these websites seem to try to obfuscate what they are doing. For example, the inspected URLs on `youporn.com` are listed in the JavaScript source in encoded form and decoded right before they are used. On other websites, the history-sniffing JavaScript is not statically inserted in a web page, but dynamically generated in a way that makes it hard to understand that history sniffing is occurring by just looking at the static code. We also found that many of these websites make use of a handful of third-party history-sniffing libraries. In particular, of the 46 cases of confirmed sniffing, 22 sites use history-sniffing code from `interclick.com` and 14 use history-sniffing code from `meaningtool.com`.

Figure 5 shows the JavaScript attack code exactly as found on `youporn.com`. The code creates an obfuscated list of inspected websites (line 1). We only show part of the list — the actual list had 23 entries. For each site, the code decodes the website name (line 6), creates a link to the target site on the page (lines 11–12), reads the color of the link that was

[1] Here and elsewhere in the paper we use the Alexa list as of February 1st, 2010.

```
 1:var k = { 0: "qpsoivc/dpn",
              1: "sfeuvcf/dpn", ... };
 2:var g = [];
 3:for(var m in k) {
 4:  var d = k[m];
 5:  var a = "";
 6:  for(var f = 0; f < d.length; f++) {
       a += String.fromCharCode(d.charCodeAt(f)-1)
     }
 7:  var h = false;
 8:  for(var j in {"http://":"","http://www.":""}){
 9:    var l = document.createElement("a");
10:    l.href = j + a;
11:    document.getElementById("ol").
                   appendChild(l);
12:    var e = "";
13:    if(navigator.appName.
                  indexOf("Microsoft") != -1 ){
14:      e = l.currentStyle.color
15:    } else {
16:      e = document.defaultView.
               getComputedStyle(l, null).
                  getPropertyValue("color")
17:    }
18:    if(e == "rgb(12, 34, 56)" ||
           e == "rgb(12,34,56)") { h = true }
19:  }
20:  if(h) { g.push(m) }
21:}
22:var b = (g instanceof Array)? g.join(",") : "";
23:var c = document.createElement("img");
24:c.src= "http://ol.youporn.com/blank.gif?id="+b;
25:document.getElementById("ol").appendChild(c)
```

Figure 5: Attack code as found on `youporn.com`

just created (lines 12–17), and finally tests the color (line 18). If the color indicates a visited link, the `h` variable is set to true, which in turn causes the link to be inserted in the list `g` of visited sites (line 20). This list is then flattened into a string (line 22), and an image is added to the current web page, encoding the flattened string into the `src` name of the image (lines 23–25).

The flow in Figure 5 from the color property `e` to the array of visited sites `g` is actually an *indirect* flow that passes through two conditionals (on lines 18 and 20). Our framework's ability to track such indirect flows allowed us to find this history-sniffing attack. Note however that our framework found the flow because the sites being tested had actually been previously visited (because we had already run the experiments once, and so all the top 50,000 Alexa global sites were in the history). If none of the tested sites had been visited, the `g` array would have remained empty, and no violation of the policy would have been observed, even though in fact the user's empty history would have been leaked. This example is precisely the *implicit flow* limitation that was mentioned in Section 2.3.

False-positive cases of history sniffing Of the 63 sites flagged by our framework, 17 are false positives in that a manual examination of the source code and run-time behavior did not allow us to conclude that they were real cases of history sniffing. Out of these 17 sites, 12 contain JavaScript code that is too complicated to understand. The remaining 5 sites contain a history sniffing widget from `interclick.com`, but no suspicious runtime behavior was detected by monitoring their network access. Our framework reported these sites either because they inspected style properties for purposes other than history sniffing, or because too many irrelevant values were tainted by our handling of indirect flows.

Provider	Description	Sites	Inspected URLs
`addtoany`	social	83	120.2
`infolinks`	advertisement	124	14
`kontera`	advertisement	87	11.5
other	-	129	13.4

Table 2: Characteristics of suspicious websites depending on JavaScript widget provider

A more stringent policy To investigate the possibility of history hijacking further, we also looked at all the sites that simply read the computed style of a link. This uncovered an additional 422 websites that read style properties of links, but did not send the properties out on the network. Unfortunately, because our framework does not cover all the corner cases of information flow in JavaScript (as discussed later), we cannot immediately conclude that these sites did not transfer the browser history. Even if we were certain that the style information was not sent to the network, it is still possible that the *absence* of sending data was used to reveal information about the browsing history. For example, if a site sent the browsing history only if a link was visited, then the server could have learned about certain links' not being visited without any information's being transfered from the client. Thus, to better understand the behavior of these additional websites, we inspected them in detail, and categorized them into two bins: *suspicious websites*, and *non-suspicious websites.*

Suspicious sites Of the 422 sites, 326 sites exhibit what we would categorize as suspicious behavior. In particular, these suspicious websites inspect a large number of external links, and some of these links are dynamically generated, or they are located in an invisible iframe. We found that many of them embed a JavaScript widget developed by another website that inspects the browser history systematically.

Table 2 shows how such widgets are used on the 326 sites. For each JavaScript widget, we give the name of its provider, a description of its provider, the number of sites embedding it, and the number of URLs it inspects on average over the sites on which it is embedded. The most notable is a menu widget developed by `addtoany.com` which inspects around 120 URLs on average to activate or deactivate each menu item depending on the browser history.

Non-suspicious sites The remaining 96 sites seemed non-suspicious. Of these, 77 simply inspect their own website history. The remaining 19 samples have JavaScript code that is too complicated for us to fully understand, but where the sites seem non-suspicious.

Incompleteness Our current implementation would miss information flow induced by certain browser built-in methods. For example, consider the code:

```
arr.push(z); var result = arr.join(',')
```

The value `z` is inserted into an array and then all the elements of the array are joined into a string using the the built-in method `join`. Even though we have implemented a wrapper object for arrays to track array assignments and reads, we have not yet implemented a complete set of wrappers for all built-in methods. Thus, in the above case, even though `result` should be tainted, our current engine would not discover this. It would be straightforward, although time-consuming, to create precise wrappers for all built-in methods that accurately reflect the propagation of taints. Moreover, our current implementation does not track information flow through the DOM, although recent techniques on tracking information flow through dynamic tree structures [25] could be adapted to address this limitation.

Even if our implementation perfectly tracked the taints of all values through program execution, our approach would still miss certain history hijacking attacks. For example, the attacking website can use a style sheet to set the font of visited links to be much larger than the size of unvisited links. By placing an image below a link, and using JavaScript to observe where the image is rendered, the attacker can determine whether the link is visited or not. These kinds of attacks that use layout information would currently be very hard to capture using a taint-based information flow engine. Some attacks in fact don't even use JavaScript. For example, some browsers allow the style of visited links to be customized with a background image that is specific to that link, and this background image can be located on the attacker's server. By observing which images are requested, the attacker can infer which links have been visited, without using any JavaScript.

Despite all these sources of incompleteness, our JavaScript information flow framework can still be used as a diagnostic tool to find real cases of history sniffing in the wild. By running experiments on the Alexa global top 50,000 we have found that 46 sites really do perform history sniffing, and one of these sites is in the Alexa global top 100. We have also found several sites that have suspicious behavior, even though our current tool does not allow us to conclude with full certainty that these sites transfer the browser's history.

5. EMPIRICAL STUDY OF ATTENTION TRACKING

We have also conducted an empirical study on the prevalence of keyboard/mouse tracking on popular websites. JavaScript code can install handlers for events related to the mouse and keyboard to collect detailed information about what a user is doing on a given website. This information can then be transfered over the network. It is not enough to take a naive approach of simply prohibiting information from being transfered into the network while the event handler is being executed since the gathered information can be accumulated in a global variable, and then sent over the network in bulk (which is what most attacks actually do).

Policies To use our information flow framework for detecting keyboard/mouse tracking, we use the following policies in our framework:

```
at $1.isMouseOver() inject "secret"
at $1.isClick() inject "secret"
at $1.isScroll() inject "secret"
...
at document.send($1, $2) block "secret" on $2
```

Benchmarks and Summary of Results We ran our information flow framework using the above policies on the front pages of the Alexa global top 1,300 websites. One of the challenges in performing this empirical study automatically is that, to observe keyboard/mouse tracking, one has to somehow simulate keyboard and mouse activity. Instead of actually simulating a keyboard and mouse, we instead chose to automatically call event handlers that have been registered for any events related to the keyboard or mouse (click,mousemove,mouseover,mouseout,scroll,copy,select). To this end, in each web page we included a common piece of JavaScript code that automatically traverses the DOM tree of the current page and systematically triggers each handler with an event object that is appropriately synthesized for the handler. Another challenge is that many of the sites that track keyboard/mouse activity accumulate information locally, and then send the information in bulk back to the server at regular intervals, using timer events. These timer events are sometimes set to intervals spanning several minutes, and waiting several minutes per site to observe any flow would drastically increase the amount of time needed to run our test suite. Furthermore, it's also hard to know, a priori, how long to wait. To sidestep these issues, in addition to calling keyboard and mouse event handlers, we also automatically call timer event handlers. We successfully ran our framework on the Alexa top 1,300 websites in a total of about two hours.

Overall, we found 328 websites on which network transfers were flagged as transferring keyboard/mouse information to the network. Of these transfers, however, many are visually obvious to the user. In particular, many websites use mouse-over events to change the appearance of the item being moused-over. As an example, it is common for a website to display a different image when the mouse moves over a thumbnail (possibly displaying a larger version of the thumbnail). Although these kinds of flows can be used to track mouse activity, they are less worrisome because the user sees a change to the web page when the mouse movement occurs, and so there is a hint that something is being sent to the server.

Ideally, we would like to focus on *covert* keyboard/mouse tracking, in which the user's activities are being tracked without any visual cues that this is happening (as opposed to *visible* tracking where there is some visual cue).[2] However, automatically detecting covert tracking is challenging because it would require knowing if the keyboard/mouse activity is causing visual changes to the web page. Instead, we used a simple heuristic that we developed after observing a handful of sites that perform *visible* keyboard/mouse tracking. In particular, we observed that when the mouse/keyboard information is sent to the server because of a visual change, the server responds with a relatively large amount of information (for example a new image). On the other hand, we hypothesized that in *covert* tracking, the server would not respond with any substantial amount of data (if any at all). As a result, of all the network transfers found by our information flow tool, we filtered out those where the response was larger than 100 bytes (with the assumption that such flows are likely to be visible tracking). After this filtering, we were left with only 115 websites. We sampled the top 10 ranked websites among these 115 sites.

Real cases of covert tracking Of the 10 sites we sampled,

[2] One could view this heuristic as *charging* sites, in bandwidth, for the privilege of exfiltrating user attention data.

Rank	Site	Description	Events
3	`youtube`	contents	click
11	`yahoo.co.jp`	portal	click
15	`sina.com.cn`	portal	click
19	`microsoft`	software	click,mouseover
34	`mail.ru`	email	click
53	`soso`	search engine	click
65	`about`	search engine	click

Table 3: Top 7 websites that perform real behavior sniffing

Rank	Site	Description	Events
503	`thesun.co.uk`	news	copy, mouseover
548	`metrolyrics`	music	copy, mouseover
560	`perezhilton`	entertainment	copy, mouseover
622	`wired`	news	copy, mouseover
713	`suite101`	blog	copy, mouseover
910	`technorati`	blog	copy, mouseover
1236	`answerbag`	search engine	copy, mouseover

Table 4: Websites that perform real behavior sniffing using `tynt.com`

we found that 7 actually perform covert keyboard and mouse tracking that we were able to reliably replicate. These 7 websites are listed in Table 3. For each site, we give its Alexa rank, its URL, a short description, and events being tracked covertly. One may be surprised to see "clicking" as being tracked covertly. After all, when a user clicks on a link, there is a clear visual cue that information is being sent over the network – the target of the link will know that the user has clicked. However, when we list clicking as being tracked covertly, we mean that there is an additional event-handler that tracks the click, and sends information about the click to another server. `google` is known for doing this: when a user clicks on a link on the search page, the click is recorded by `google` through an event handler, without any visual cue that this is happening (we do not list `google` in Table 3 because we only visit the front pages of websites, and `google`'s tracking occurs on the search results page)

The most notable example in Table 3 is the `microsoft.com` site, which covertly tracks clicking and mouse behavior over many links on the front page and sends the information to the web statistics site `webtrends.com`.

Cases of visible tracking Of the 10 sites that were sampled, 3 were actually cases of visible tracking, despite our filtering heuristic. In one of these cases, the server responded with very small images (less than 100 bytes) that were being redrawn in response to mouse-over events. In an other case, the server responded with small JSON commands that caused some of the web page to be redrawn. In all of these cases, there was a clear visual cue that the information was being sent to the server.

Cases of using tracking libraries Of the 115 sites on which the filtered flows were reported, we found that 7 used a behavior tracking software product developed by `tynt.com` to track what is copied off the sites. These 7 websites are listed in Table 4. The library monitors the copy event. When a visitor copies the content of a web page to her clipboard, the library inserts the URL of the page into the copied content. Thus, the URL is contained within subsequent pastes from the clipboard, *e.g.,* in emails containing the pasted text, thereby driving more traffic to the URL. Using our framework, we discovered that on each client website, the copied content is also transferred to `tynt.com`.

Suspicious website While investigating several sites that installed event handlers, we also found that the `huffingtonpost.com` site exhibits suspicious behavior. In particular, every article on the site's front page has an on-mouse-over event handler. These handlers collect in a global data structure information about what articles the mouse passes over. Despite the fact the information is never sent on the network, we still consider this case to be suspicious because not only is the infrastructure present, but it in fact collects the information locally.

6. RELATED WORK

Information flow [7] and non-interference [11] have been used to formalize fine-grained isolation for nearly three decades. Several *static* techniques guarantee that certain kinds of inputs do no flow into certain outputs. These include type systems [31, 23], model checking [28], Hoare-logics [1], and dataflow analyses [18, 26]. Of these, the most expressive policies are captured by the dependent type system of [21], which allows the specification and (mostly) static enforcement of rich flow and access control policies including the dynamic creation of principals and declassification of high-security information. Unfortunately, fully static techniques are not applicable in our setting, as parts of the code only become available (for analysis) at run time, and as they often rely on the presence of underlying program structure (*e.g.,* a static type system).

Several authors have investigated the use of dynamic taint propagation and checking, using specialized hardware [27, 29], virtual machines [4], binary rewriting [22], and source-level rewriting [5, 19]. In the late nineties, the JavaScript engine in Netscape 3.0 implemented a Data Tainting module [10], that tracked a single taint bit on different pieces of data. The module was abandoned in favor of signed scripts (which today are rarely used in Web 2.0 applications), in part because it led to too many alerts. Our results show that, due to the prevalence of privacy-violating flows in popular Web 2.0 applications, the question of designing efficient, flexible and usable flow control mechanisms should be revisited. Recently, Vogt et al. [30] modified the browser's JavaScript engine to track a taint bit that determines whether a piece of data is sensitive and report an XSS attack if this data is sent to a domain other than the page's domain, and Dhawan and Ganapathy [8] used similar techniques to analyze confidentiality properties of JavaScript browser extensions for Firefox. Our approach provides a different point in the design space. In particular, our policies are more expressive, in that our framework can handle both integrity and confidentiality policies, and more fine-grained, in that our framework can carry multiple taints from different sources at the same time, rather than just a single bit of taint. On the downside, our approach is implemented using a JavaScript rewriting strategy rather than modifying the JavaScript run-time, which results in a larger performance overhead. Dynamic rewriting approaches for client-side JavaScript information flow have also been investigated in a theoretical setting [5, 19]. Our work distinguishes itself from these more theoretical advances in terms of experimental evaluation: we have focused on implementing a rewriting-based approach that works on a large number of popular sites, and on evaluating the prevalence of privacy-violating flows on these websites.

One way to ensure safety on the *client* is to disallow unknown scripts from executing [16]. However, this will likely make it hard to use dynamic third-party content. Finally, Yu et al. [32] present a formal semantics of the interaction between JavaScript and browsers and builds on it a proxy-based rewriting framework for dynamically enforcing automata-based security policies [17]. These policies are quite different from information flow in that they require sparser instrumentation, and cannot enforce fine-grained isolation.

The possibility of history sniffing was first raised in the academic community a decade ago [9]. The original form of history sniffing used timing difference between retrieving a resource that is cached (because it has previously been retrieved) and one that is not. In general, many other forms of history sniffing are possible based on CSS link decoration, some of which (for example, setting the background property of a visited link to `url(...)`) work even when JavaScript is disabled. This, together with the genuine user-interface utility that visited link decoration provides, is the reason that history sniffing is so difficult to address comprehensively in browsers (cf. [13] for a proposed server-side solution, [12] for a proposed client-side solution and [2] for the fix recently deployed by the Firefox browser.) The potential of history sniffing has been recently proven to be enormous [14]. However, since to date there has been no public disclosure regarding the use of history sniffing, and no publicly available tools for detecting it, we expect that, today, many malicious sites will prefer the simple, robust approach of querying and exfiltrating link computed style. Accordingly, it is this data flow that we focus on; if there are sites that use other approaches, we will not have detected them. Our goal in this paper is to draw attention to the use of clandestine history sniffing at popular, high-traffic sites, which means that false negatives are acceptable. In future work, we hope to extend our tool to detect other forms of history sniffing as well.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a rewriting-based information flow framework for JavaScript and evaluated the performance of an instantiation of the framework. Our evaluation showed that the performance of our rewriting-based information flow control is acceptable given our engineering and optimization efforts, but it still imposes a perceptible running-time overhead. We also presented an extensive empirical study of the prevalence of privacy-violation information flows: cookie stealing, location hijacking, history sniffing, and behavior tracking. Our JavaScript information flow framework found many interesting privacy-violating information flows including 43 cases of real history sniffing over the Alexa global top 50,000 websites, despite some incompleteness.

One direction for future work is a larger scale study on privacy-violating information flows. Such a study could perform a *deeper* crawl of the web, going beyond the front-pages of web sites, and could look at *more* kinds of privacy-violating information flows. Moreover, we would also like to investigate the prevalence of security attacks led by privacy-violating information flows like phishing and request forgery.

Another direction for future work is to extend our current framework to become a bullet-proof client-side *protection* mechanism. The primary purpose of our tool so far has been to *observe* existing flows in the wild, a scenario for which we don't need to worry about malicious code trying to circumvent our system. However, with additional work, our framework could possibly lead to a *protection* mechanism as well. For this purpose, we would have to soundly cover all possibly forms of information flow, including implicit flow, flows induced by the DOM and browser built-in APIs. In addition, we would also need better performance to deliver a practical browsing experience. However, we believe that with careful and extensive engineering efforts, there is a possibility that our framework could lead to a practical protection mechanism.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0644306, CCF-0644361, CNS-0720802, CNS-0831532, and CNS-0964702. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *Proceedings of SAS 2004*, volume 3148 of *LNCS*, pages 100–15. Springer-Verlag, Aug. 2004.

[2] L. D. Baron. Preventing attacks on a user's history through CSS :visited selectors, Apr. 2010. Online: `http://dbaron.org/mozilla/visited-privacy`.

[3] Bugzilla@Mozilla. Bug 147777 – :visited support allows queries into global history, May 2002. Online: `https://bugzilla.mozilla.org/show_bug.cgi?id=147777`.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In M. Blaze, editor, *Proceedings of USENIX Security 2004*, pages 321–36. USENIX, Aug. 2004.

[5] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In M. Backes and A. Myers, editors, *Proceedings of CSF 2010*. IEEE Computer Society, July 2010.

[6] A. Clover. Timing attacks on Web privacy. Online: `http://www.securiteam.com/securityreviews/5GP020A6LG.html`, Feb. 2002.

[7] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[8] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In C. Payne and M. Franz, editors, *Proceedings of ACSAC 2009*, pages 382–91. IEEE Computer Society, Dec. 2009.

[9] E. W. Felten and M. A. Schneider. Timing attacks on Web privacy. In S. Jajodia, editor, *Proceedings of CCS 2000*, pages 25–32. ACM Press, Nov. 2000.

[10] D. Flannagan. *JavaScript: The Definitive Guide.* O'Reilly, fifth edition, Aug. 2006.

[11] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of IEEE Security and Privacy ("Oakland") 1982*, pages 11–20. IEEE Computer Society, Apr. 1982.

[12] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from Web privacy attacks. In C. Goble and M. Dahlin, editors, *Proceedings of WWW 2006*, pages 737–44. ACM Press, May 2006.

[13] M. Jakobsson and S. Stamm. Invasive browser sniffing and countermeasures. In C. Goble and M. Dahlin, editors, *Proceedings of WWW 2006*, pages 523–32. ACM Press, May 2006.

[14] A. Janc and L. Olejnik. Feasibility and real-world implications of Web browser history detection. In C. Jackson, editor, *Proceedings of W2SP 2010*. IEEE Computer Society, May 2010.

[15] D. Jang, R. Jhala, S. Lerner, and H. Shacham. Rewriting-based dynamic information flow for JavaScript. Technical report, University of California, San Diego, Jan. 2010. Online: `http://pho.ucsd.edu/rjhala/dif.pdf`.

[16] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In P. Patel-Schneider and P. Shenoy, editors, *Proceedings of WWW 2007*, pages 601–10. ACM Press, May 2007.

[17] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript instrumentation in practice. In G. Ramalingam, editor, *Proceedings of APLAS 2008*, volume 5356 of *LNCS*, pages 326–41. Springer-Verlag, Dec. 2008.

[18] M. S. Lam, M. Martin, V. B. Livshits, and J. Whaley. Securing Web applications with static and dynamic information flow tracking. In R. Glück and O. de Moor, editors, *Proceedings of PEPM 2008*, pages 3–12. ACM Press, Jan. 2008.

[19] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In K. Rannenberg and V. Varadharajan, editors, *Proceedings of SEC 2010*, Sept. 2010.

[20] L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of IEEE Security and Privacy ("Oakland") 2010*, pages 481–496. IEEE Computer Society, 2010.

[21] A. C. Myers. Programming with explicit security policies. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *LNCS*, pages 1–4. Springer-Verlag, Apr. 2005.

[22] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In D. Boneh and D. Simon, editors, *Proceedings of NDSS 2005*. ISOC, Feb. 2005.

[23] F. Pottier and V. Simonet. Information flow inference for ML. In J. C. Mitchell, editor, *Proceedings of POPL 2002*, pages 319–330. ACM Press, Jan. 2002.

[24] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of Web-based malware. In N. Provos, editor, *Proceedings of HotBots 2007*. USENIX, Apr. 2007.

[25] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In M. Backes and P. Ning, editors, *Proceedings of ESORICS 2009*, volume 5789 of *LNCS*, pages 86–103. Springer-Verlag, Sept. 2009.

[26] U. Shankar, K. Talwar, J. S. Foster, , and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In D. Wallach, editor, *Proceedings of USENIX Security 2001*, pages 201–17. USENIX, Aug. 2001.

[27] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In K. McKinley, editor, *Proceedings of ASPLOS 2004*, pages 85–96. ACM Press, Oct. 2004.

[28] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin, editor, *Proceedings of SAS 2005*, volume 3672 of *LNCS*, pages 352–67. Springer-Verlag, Sept. 2005.

[29] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In A. González and J. P. Shen, editors, *Proceedings of MICRO 2004*, pages 243–54. IEEE Computer Society, Dec. 2004.

[30] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In W. Arbaugh and C. Cowan, editors, *Proceedings of NDSS 2007*. ISOC, Feb. 2007.

[31] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In T. Reps, editor, *Proceedings of POPL 2000*, pages 268–76. ACM Press, Jan. 2000.

[32] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In M. Felleisen, editor, *Proceedings of POPL 2007*, pages 237–49. ACM Press, Jan. 2007.