Rewriting-based Dynamic Information Flow for JavaScript [*]

Dongseok Jang
UC San Diego
d1jang@cs.ucsd.edu

Ranjit Jhala
UC San Diego
jhala@cs.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Abstract

JavaScript web applications often dynamically load third-party code, which in some cases can steal or corrupt important client information. In this paper, we present a rewriting-based approach for enforcing confidentiality and integrity policies that respectively specify what information can flow into and from untrusted third-party code. We have implemented our approach in the Chrome browser, and we present experiments that evaluate the efficiency and precision of our technique on real-world websites.

1. Introduction

JavaScript is a dynamically typed language that can be embedded in web pages and executed by the web browser. JavaScript is becoming the lingua franca of modern Web 2.0 applications. Almost every popular web site uses some amount JavaScript, and many interactive web sites, like search engines, email sites and mapping applications are almost entirely implemented in client-side JavaScript.

Although JavaScript has enabled web developers to provide a richer web experience, JavaScript has also opened up the possibility for a variety of security vulnerabilities. In particular, typical JavaScript applications are made up from code originating from many different sites. Unfortunately, JavaScript does not provide strong protection mechanisms, so that code included from a particular site, say for displaying an ad, essentially runs in the context of the hosting page. Thus, the ad code has access to all the information on the hosting web page, including the cookie, the location bar, and any other information stored on the page. The lack of strong protection mechanisms in JavaScript has lead to a variety of attacks like cross-site scripting and cross-site request forgery.

To make JavaScript more secure, ideally we would like to specify and enforce confidentiality policies stating what parts of the web page can be read by what JavaScript code and integrity policies stating what JavaScript code can affect what parts of the page. One formalism that is well suited for expressing these kinds of policies is *information flow policies* which specify where in the code a given value can flow to. Thus, for example, a web user could state using an information flow policy that sensitive information stored in a cookie should not flow to any code loaded from third party ad servers.

Although there has been work on static enforcement of information flow for a variety of languages, performing static information flow on a language like JavaScript is extremely hard. JavaScript has many features that make precise static analysis all but impossible. These features include dynamic loading of code from the network, dynamic code construction and evaluation, prototypes and dynamic dispatch, dynamically added and removed fields, and dynamic field assignments (where the field name is constructed at runtime).

Rather than try to analyze the JavaScript code statically, in this paper we present a framework that tracks information flow for JavaScript *dynamically*. Our framework inserts and propagates taints through the program as it runs to enforce confidentiality and integrity policies. The dynamic nature of our analysis allows it to precisely track flow even through the many features of JavaScript that make static analysis hard.

One approach to implementing dynamic information flow for JavaScript is to modify the JavaScript runtime inside the browser so that it inserts and propagates taints. Such an approach, however, would require understanding all of the details of how the JavaScript runtime works, which is a daunting task in modern web browsers that use sophisticated JIT-based runtimes. Instead, our framework works by rewriting the JavaScript code so that it propagates taints by itself. Although the rewriting is performed inside the browser, implementing our approach only requires understanding the browser's AST data structure, and none of the complexity of the JavaScript run-time.

Tracking information flow using rewriting poses several challenges. First, the rewriting must be performed in such a way that all JavaScript code, even code that is dynamically loaded and executed, gets rewritten. Second, one has to come up with a way to track taint information for *all* values, including *unboxed values*. Unboxed values are particularly challenging as they represent primitive values that cannot be augmented with fields to store the taints, and as they do not have addresses that can be used to look up the taints in a table. This problem is made worse by the fact that some unboxed values *cannot* be boxed without breaking the program. Third, one has to perform the rewriting in a way that preserves the semantics of the JavaScript code, which is made difficult by features like DOM accesses, interactions with native libraries inside the browser, and appropriate boxing and unboxing of values. In this paper, we make the following contributions:

- We present a rewrite-based dynamic information flow framework for JavaScript. Section 2 gives an overview of our framework, whereas Sections 3 and 4 provides a detailed description of the rewrite rules.
- We have implemented an instantiation of our framework in the Chrome browser, and have used our enhanced browser to surf the web on on a variety of real web sites. Although our framework slows the browsing experience, all sites are still usable.
- We have used our Chrome implementation to measure the precision and performance of our framework on the Alexa top 100 web sites. We show that

2. Overview

We begin with an example that gives a high-level overview of our approach for dynamically enforcing information flow policies.

[*] This work was supported by NSF CAREER grants CCF-0644306, CCF-0644361, NSF PDOS grant CNS-0720802, NSF Collaborative grant CCF-0702603.

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

initSettings("a.com");

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

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

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

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

Figure 2. A malicious string from `adserver.com`

Webpage Consider the JavaScript in Figure 1. Suppose that this code is a distillation of the JavaScript on a webpage belonging to the domain `a.com`. The webpage has a text box whose contents are stored in the global variable `searchBoxValue`. 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 webpage with an advertisement tailored to the particular user.

Malicious Code Suppose that for one reason or another , the string returned by the call to `adserver.com` was that shown in Figure 2. 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 Information Flow

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 [5, 11], 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 We require that website developers specify fine-grained isolation via information flow policies. An *object-location* is a pair of an object identifier and a field name. For example the object-location $($`document`, "`location`"$)$ describes the object-location corresponding to the loaded website's URL. An *integrity* policy is a map from object-locations to URLs whose code is allowed to *influence* the values stored at that object-location. For example, the developer for `a.com` would specify an integrity policy that mapped $($`document`, "`location`"$)$ to `a.com`, indicating that only values from `a.com` can flow into the document's location. By default, if a object-location is not mapped to anything, we assume that it can be influenced by values originating at *any* URL. A *confidentiality* policy is a map from object-locations to URLs whose code is allowed to *be influenced by* the values stored at the object-location. For example, the developer for `a.com` would specify a confidentiality policy that mapped $($`document`, "`cookie`"$)$ to `a.com`, indicating that the values held in the document's cookie can only flow into code that originates at `a.com`.

Enforcement Policies are enforced via a three-step process.

1. **Taint Injection:** First, we associate with each program object a set of *taints* that describe where the object has come from, and where it is allowed to go. An *integrity taint* is of the form $\langle \mathit{InfBy}, \mathit{url}\rangle$ that specifies that the tainted object has been *influenced by* code originating at url. Every object created in code from url is injected with this taint. A *confidentiality taint* is of the form $\langle \mathit{CnfTo}, \mathit{url}\rangle$ that specifies that the tainted object is *confidential to* code originating at url. Every value stored in a confidential object-location is injected with this taint.
2. **Taint Propagation** Second, we propagate the taints with the objects as they flow through the program via assignments, procedure calls *etc.*. A static analysis would carry out this propagation via some form of dataflow analysis [14], while a dynamic analysis [4] would attach taints with the objects and copy them around together with the object.
3. **Taint Checking** Third, at each point where value-flow happens, *i.e.,* at each assignment of a source object to a target object-location, we check if the assignment is legal with respect to the taints associated with the objects. The flow is legal if: (a) for each integrity taint $\langle \mathit{InfBy}, \mathit{url}\rangle$ carried by the object, the target object-location is allowed to be influenced by url, (b) for each confidentiality taint $\langle \mathit{CnfTo}, \mathit{url}\rangle$ carried by the object, the assignment occurs within code originating at url.

2.2 Rewriting-based Dynamic Information Flow

While the above three-step recipe for policy enforcement can be implemented statically or dynamically, the nature of JavaScript and dynamic code loading makes precise static enforcement problematic, as it is impossible to predict what code will be loaded at runtime. 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.

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 webpage 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 ar-

```
//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 = document.searchBoxValue;

  //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, document, "location", "a.com"),
  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 3. Rewritten code from a.com. The comments above each block denote the original version of the rewritten block.

guments of eval within a call to RW to ensure they are (recursively) rewritten before evaluation [21].

When the rewriting procedure is invoked on the code from Figure 1 and the URL a.com, it emits the code shown in Figure 3. The rewriting procedure rewrites each statement and expression. (In Figure 3, 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 propagate taints, we extend *every* object with a special field taint. Whenever an object is created. 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 taint field is set to $\langle \mathit{InfBy}, \mathit{url} \rangle$ indicating that the object has an integrity taint denoting where it was created. 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.*.

Checking To check taints, we add, before *every* assignment, a call to a function check which takes four arguments and determines whether the value of its first argument can be assigned to (*i.e.,* is allowed to flow to) the object-location corresponding to its second and third arguments, in code from the URL denoted by the fourth argument. For example, consider the rewritten version of the assignment to document.location in the body of doSearch. The rewrite inserts before the assignment, a call to check that determines whether the taints stored in searchQry.taint allow it to be assigned to document.location in code from a.com. At run-time, when this call is executed it halts the program with a flow-violation message if searchQry.taint has a taint of the form $\langle \mathit{InfBy}, \mathit{url} \rangle$ where *url is not* a.com, the only URL that is allowed (by the integrity policy) to influence document.location.

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 a variable (which has already been 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 x.taint and $\langle \mathit{InfBy}, \texttt{url} \rangle$ to the taint-set. Second, TSET.boxAndTaint(x, url), which creates a boxed version of x (if it is not boxed), taints it with $\langle \mathit{InfBy}, \texttt{url} \rangle$ 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 $\langle \mathit{InfBy}, \text{“a.com”} \rangle$, 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.

Assumptions The above code makes two assumptions for ease of exposition. First, the fields taint and url are not read, written or removed by any code other than was placed for tracking. Second, that eval fails on programmatically constructed strings which lack a url field. In practice, we use the $\langle \mathit{InfBy}, \cdot \rangle$ taints associated with the string to determine the URL from which it originated.

Attack Prevention Suppose that the load(·) operation returns the string from Figure 2. The rewritten code invokes the rewriting function RW on the string, and the URL adserver.com yielding the string shown in Figure 4. Notice that now, the argument passed to

```
tmp10 = box("evil.com", "adserver.com"),
initSettings(tmp10);
```

Figure 4. Rewritten version of code from Figure 2.

`initSettings` carries the taint $\langle InfBy$, "`adserver.com`"$\rangle$, 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 $\langle CnfTo, url\rangle$ taint to the results of each *read* from a object-location confidential to url. To *check* taints, the `TSET.boxAndTaint(x,url)` function creates a boxed-and-tainted version of `x` and *checks* that the taints indicate that the value can be *read by* `url`. If so, the tainted version is returned, and if not, execution is halted with a warning.

The above example gives a high-level overview of our rewriting-based approach to enforcing information flow policies. Next, we describe a core language (Section 3), and then precisely describe the rewriting rules for the core language (Section 4).

3. Core JavaScript

We start by formalizing Core JavaScript, a subset of the complete language which we use to describe our rewriting-based approach.

3.1 Language: Core JavaScript

For ease of exposition, we assume that a Core JavaScript program is an expression (*i.e.,* unlike JavaScript we do not distinguish between statements and expressions). Core JavaScript expressions include:

basic constants of the form c that represent integers, strings *etc.*,

variable reads of the form x

field reads of the form $e_1[e_2]$, where e_1 is an expression that evaluates to the object whose field is being read, and e_2 is an expression that evaluates to the name of the field being read

binary operations of the form e_1 `op` e_2 that include primitive operations like integer addition, string concatenation *etc.*,

object literals of the form $\{\texttt{f}_1\!:\!e_1\ldots\}$ that map a set of fields $\texttt{f}_1\ldots$ to a set of objects represented by $e_1\ldots$ respectively

variable assignments of the form $x = e$; the assignment updates x and evaluates to the object that e evaluates to,

field assignments of the form $e_1[e_2] = e_3$, where e_1 evaluates to the object whose field is updated, e_2 evaluates to (a string naming) the field being written, and e_3 is the expression whose value the field is updated with; field-assignments evaluate to the object that e_3 evaluates to

sequence expressions of the form $e_1; e_2$ where e_1 is evaluated first, then e_2; the result is the value of e_2

branches of the form `if` $e_1\ e_2\ e_3$; a branch expression evaluates to the trivial null object

functions of the form `fun`$(x_1\ldots)\{e\}$ where $x_1\ldots$ are the formals of the function and e the function's body (the function returns the value of e); in our encoding, methods are functions with a `this` parameter, that are bound to the fields of objects

function calls of the form $e(e_1\ldots)$ where e evaluates to the callee and $e_1\ldots$ to the arguments; we encode method calls as function calls made through a field, and for which the target object is passed as the first parameter (for example `x.f(x, ...)`)

with-expressions of the form `with` (e_1) $\{e_2\}$ where e_1 evaluates to an object *within whose scope* e_2 is evaluated; *i.e.,* variable accesses in e_2 correspond to field on e_1

Dynamic Loading Core JavaScript includes two operations that are used to model dynamic code loading:

loads of the form `load`(e) where e evaluates to a URL (string); the contents of the page identified by the URL is retrieved and returned as a string

evals of the form `eval`(e) where e evaluates to a string that is dynamically executed at that point

Example Suppose that we wish to load and run a web-application from the URL `a.com`. We model this by the program:

```
eval(load("a.com"))
```

If the code from `a.com` requires a function `foo` from a library provided by `b.com` then the above load could return the string:

```
eval(load("b.com")); foo(x);...
```

where the first line loads the code from the library site, and the second line calls the library function. The code loaded from `b.com` could contain its own loads and evals in order to dynamically stitch together code from other libraries. The `eval` operation can also be used on strings that have been constructed locally (*e.g.,* by concatenating substrings).

DOM Instead of explicitly modeling the webpage and the DOM, we assume that there is a "global" variable called `document` with the appropriate fields (*e.g.,* `cookie`, `location`) that are manipulated in the Core JavaScript program.

Operational Semantics The operational semantics of Core JavaScript operations are based on the semantics of the corresponding operations for JavaScript [10].

4. Information Flow via Rewriting

Our goal is to *track* the flow of information through the program, in order to *prevent* flows that violate confidentiality and integrity policies. Next, we formalize the notion of policies, and describe how we rewrite the program so that it injects, propagates and checks taints in order to enforce flow policies.

4.1 Information Flow Policies

First, we formalize the notion of object-locations and confidentiality and integrity policies.

Object-locations Suppose that every object that is created in the program is stamped with a unique object identifier drawn from the set ID. A *object-location* is a pair of an object identifier and a *field name* (string). For example, the top-level DOM variable `document` refers to a special object identified by $id_{\texttt{document}}$. Hence, $(id_{\texttt{document}}$, "`cookie`"$)$ is the object-location corresponding to the top-level document's cookie, and $(id_{\texttt{document}}$, "`location`"$)$ is the object-location corresponding to the top-level document's URL.

Policies A *policy* is a pair of the form $(\mathsf{Cnf}, \mathsf{Itg})$, where Cnf is a *confidentiality* policy that is a *finite* map from object-locations to the URLs that are allowed to *read* that object-location, Itg is a *integrity* policy that is a *finite* map from object-locations to the URLs that are allowed to *write* that object-location.

Example The policy $(\mathsf{Cnf}, \mathsf{Itg})$ where

$$\mathsf{Cnf} \doteq [(id_{\texttt{document}}, \text{“}\texttt{cookie}\text{”}) \mapsto \text{“}\texttt{a.com}\text{”}]$$
$$\mathsf{Itg} \doteq [(id_{\texttt{document}}, \text{“}\texttt{location}\text{”}) \mapsto \text{“}\texttt{a.com}\text{”}]$$

specifies that the value of `document.cookie` (resp. `document.location`) should only flow to (resp. from) code originating at `a.com`.

4.2 Rewriting Algorithm

Figure 5 summarizes our rewriting procedure $\mathsf{RW}(e, \texttt{url})$, which takes as input a Core JavaScript program e and the URL the expression was loaded from `url` and returns a rewritten version of the program. The figure is given as a series of rules of the form:

$$\mathsf{RWA}(e, \texttt{url}) \doteq e'$$

where each rule describes how an expression that matches e is rewritten to the expression e'. We use *italics* to denote meta variables that range over expressions and `typewriter` to denote concrete expressions and variables.

URL Tracking The `url` is used to determine the domain from which the code was dynamically loaded. In particular, the rewriting procedure uses the `url` to (1) place *checks* that determine whether a given confidential value may flow into code from `url`, (2) place new $\langle \mathit{InfBy}, \cdot \rangle$ *taints* on values created by the loaded code, in order to subsequently prevent those values from flowing into and violating the integrity of a object-location.

Dynamic Rewriting As dynamically loaded code can itself load code dynamically, our rewriting procedure places a call to *itself* when rewriting an `eval` expression. These deferred calls are signified via the typewriter (`RW`) font, in contrast to the serif (RW) font which denotes the “eager” invocations that rewrites (known) subexpressions.

Goals At a high level, our rewriting has two goals. First, to ensure that each object has a special `taint` field that contains taints which describe the URLs that the object can be read by, and has been influenced with. Second, to ensure that each assignment is checked to determine if it violates the given flow policy. Next, we describe how the rewriting procedure achieves these goals by describing how it tracks and checks different kinds of flows.

4.3 Direct Flows

We start by describing how our rewriting tracks *direct* flows, where the value of one object-location flows into another due to a sequence of assignments.

Taint-Set Library Our dynamic taint tracking is carried out using a *taint-set object* stored in a variable called `TSET`. The `TSET` object implements a variety of methods that are invoked to store and access taints as the program executes. We define a `TSET` variable for each function, instead of a single global `TSET` variable as it simplifies the handling of closures, and is safe with respect to the asynchronous concurrency supported by JavaScript (which we do not model in our subset for simplicity). Instead of describing all the `TSET` operations at once, we describe how the operations are invoked at different points in the rewritten program.

Direct Taint Stack (`TSET.direct`) To deal with the complex rules that govern the order of evaluation of sub-expressions, and the fact that (like C) JavaScript assignments return a value (corresponding to the value being assigned), we carry out a form of dynamic three-address translation, by equipping the `TSET` object with a *direct-taint stack* (named `TSET.direct`). This object contains a stack of taint-sets, and the rewriting maintains the invariant that at each point where an (original) sub-expression finishes evaluating, the taints associated with that sub-expression's value are stored at the top of the direct-taint stack. That is, our rewriting function has the property that when e is rewritten to e', the value that e' evaluates to is exactly the same as e (except for the `taint` field) and the taints of e (and hence, e') are stored at the top of the direct-taint stack. We ensure this invariant by having the rewrite function RW call an auxiliary function RWA to actually carry out the rewrite, after which the result is *wrapped* inside a call to `TSET.direct.add` which adds the result's taints and the taint $\langle \mathit{InfBy}, \texttt{url} \rangle$ to the top of the direct-taint stack and returns the result.

Confidentiality Enforcement is carried out by the method `TSET.boxAndTaint`, which takes as input two parameters `x`, an object and `url` a URL in which a read occurs. It creates a boxed version of the object (if the object is unboxed) and adds to the object's `taint` field (1) *all* the taints currently accumulated in the top of the taint-set stack `TSET`, and (2) *all* the taints due to control dependences, accumulated in the indirect-taint stack (described later). Before returning the boxed-and-tainted object, `TSET.boxAndTaint` checks if the returned object has a confidentiality taint of the form $\langle \mathit{CnfTo}, \mathit{url} \rangle$ where $\mathit{url} \neq \texttt{url}$. If such a taint exists, it means that code from `url` is not allowed to read `x` and so execution is halted.

Integrity Enforcement is carried out by the method `TSET.check` which takes as input a source object `x`, a target object-location represented by a pair of an object `d_obj` and field name `d_fld`. The method checks if `x` has an integrity taint of the form $\langle \mathit{InfBy}, \mathit{url} \rangle$ where $\mathit{url} \neq \mathsf{Itg}(\texttt{d\_obj}, \texttt{d\_fld})$. If such a taint exists, it means that `x` has been influenced by code from URLs that should not influence the target object-location and so execution is halted.

Taint Accumulation The base cases for the rewriting procedure are constants c and variable reads x, which are simply rewritten to themselves (they will be boxed-and-tainted before being assigned anywhere). The latter is simply rewritten to itself. For binary operations, we recursively invoke the rewriting procedure to create new temporary variables to store the values of each sub-expression, and then apply the binary operation to the temporaries. Note that the recursive calls have the effect of *accumulating* the taints of the two sub-expressions on the top of the direct-taint stack. Thus, the taints for the result, which is the union of the taints for the sub-expressions, is at the top of the direct-taint stack when the rewritten expression finishes evaluating.

Assignments For variable assignment expressions $x = e$, the rewriting procedure first rewrites the right-hand side (RHS) e. The rewritten RHS is wrapped in a call to `TSET.boxAndTaint` which returns the object together with a `taint` field that has been embellished with *all* the associated taints for that expression including both *direct* taints from sub-expressions and *indirect* taints from the control-dependences under which the assignment occurs. The call also checks that code from `url` is allowed to read the RHS value, and if not, halts execution. The returned boxed-and-tainted object is stored in a temporary variable. Next, the rewriting adds a call to `TSET.check` which enforces the integrity policy by checking whether an object with the RHS taints is allowed to flow into the object-location represented by x. If the check fails, the program halts with an error message. Otherwise, the tainted object is assigned to x and returned. (The $\leftarrow$ denotes box-sensitive assignment, a notion we explain later: for now, think of it as plain assignment.) The rewriting for field-assignments is similar, except that in this case, to identify the object-location being assigned to, the `TSET.check` is passed the host-object $\texttt{tmp}_1$ and the field $\texttt{tmp}_2$.

Field Reads For field read expressions of the form $e_1[e_2]$, we rewrite the host object e_1 and field name expressions e_2 and store the results in new temporaries $\texttt{tmp}_1$ and $\texttt{tmp}_2$ respectively. Finally, the rewritten expression uses the temporaries to carry out the field

```
RW(e, url) ≐
    tmp = RWA(e, url)
    TSET.direct.add(tmp, url)

RWA(load(e), url) ≐
    tmp = RW(e, url),
    tmp_1 = load(tmp),
    tmp_1["url"] = tmp,
    tmp_1

RWA(eval(e), url) ≐
    tmp = RW(e, url),
    eval(RW(tmp, tmp["url"]))

RWA(c, url) ≐ c

RWA(x, url) ≐ x

RWA(e_1 op e_2, url) ≐
    tmp_1 = RW(e_1, url),
    tmp_2 = RW(e_2, url),
    unbox(tmp_1) op unbox(tmp_2)

RWA({f_1 : e_1 ...}, url) ≐
    tmp_1 ... = RW(e_1 ..., url)
    {f_1 : tmp_1 ...}

RWA(e_1; e_2, url) ≐
    RW(e_1, url); RW(e_2, url)

RWA(x = e, url) ≐
    TSET.direct.push(),
    tmp = RW(e, url)
    tmp = TSET.boxAndTaint(tmp, url),
    TSET.check(tmp, None, "x"),
    x ← tmp
    TSET.direct.pop(),
    tmp

RWA(e_1[e_2], url) ≐
    tmp_1 = RW(e_1, url),
    tmp_2 = RW(e_2, url),
    tmp_3 = "taint_" + tmp_2,
    TSET.direct.add(tmp_1[tmp_3], url),
    tmp_1[tmp_2]

RWA(e_1[e_2] = e_3, url) ≐
    tmp_1 = RW(e_1, url),
    tmp_2 = RW(e_2, url),
    TSET.direct.push(),
    tmp_3 = RW(e_3, url),
    tmp_3 = TSET.boxAndTaint(tmp_3, url),
    TSET.check(tmp_3, tmp_1, tmp_2),
    tmp_1[tmp_2] ← tmp_3,
    TSET.direct.pop(),
    tmp_3

RWA(if e_1 e_2 e_3, url) ≐
    tmp_1 = RW(e_1, url)
    TSET.indirect.push(tmp_1),
    if (unbox(tmp_1))
        RW(e_2, url))
        RW(e_3, url)),
    TSET.indirect.pop()

RWA(fun(x_1 ...){e}, url) ≐
    tmp = TSET.indirect.get(),
    fun(x_1 ..., I){
        TSET = new TSET();
        TSET.indirect.push(tmp);
        TSET.indirect.push(I);
        RW(e, url)
    }

RWA(e(e_1 ...), url) ≐
    tmp = RW(e, url),
    RW(tmp_1 = e_1 ..., url),
    tmp(tmp_1 ..., TSET.indirect.get())

RWA(with (e_1) {e_2}, url) ≐
    tmp_1 = RW(e_1, url),
    with(tmp_1){
        TSET.with.push(tmp_1),
        tmp_2 = RW(e_2, url)
        TSET.with.pop(),
        tmp_2
    }
```

Figure 5. Rewrite Function: Each `tmp` variable is fresh, we elide the `var` declarations for clarity.

read. For now, ignore the `TSET.direct.add`$(\cdot,\cdot)$ call, we explain it when describing box-sensitive assignments.

Direct-Taint Stack Revisited Notice that the rewritten assignment expressions are enclosed within a pair of calls to `TSET.direct.push()` and `TSET.direct.pop()`. To understand why, consider the following JavaScript expression:

$$e_1 + (\mathtt{x} = e_2) + e_3$$

The result of the expression must contain the accumulated taints of e_1, e_2 and e_3, but the variable `x` should be updated with an object that only contains the taints of e_2. To achieve this, when rewriting an assignment we *push* a new, initially empty set of taints onto the top of the direct-taint stack, accumulate the taints for that assignment at the top of the stack, and when we are done with that assignment, we *pop* the taint set off the stack to resume the accumulation on the super-expression to which the assignment belonged. Thus for the example above, we would accumulate the taints for e_1 on the top of the stack, then push a new taint set on top of the stack to accumulate the taints for e_2. When the assignment is finished (and `x` assigned to the result of e_2), the taint set is popped off the stack and *added* to the taints accumulated for e_1. Finally, the taints for e_3 are computed and added to the top of the stack resulting in the accumulation of taints for the whole expression.

4.4 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`$(\cdot,\cdot)$ 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 $\mathtt{tmp}_1$, 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`$(\cdot,\cdot)$ 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 (such as the one above) but not *implicit* flows that occur due to the *absence* of an assignment. 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 [4] will fail to detect and prohibit such implicit flows.

Function Definitions For each function definition of the form $\texttt{fun}(x_1 \ldots)\{e\}$, our rewriting does several things. First, we invoke TSET.indirect.get() to save the indirect taints at the point of definition in an temporary variable tmp that is in-scope for the rewritten function. Second, we create a new formal parameter I for the rewritten function; this parameter is used to pass in the indirect taints that hold at the caller. Third, we equip the rewritten function with its own local TSET object, and we initialize the indirect taint stack with the indirect taints at the definition site (tmp) and at the callsite (I). As a result, the rewritten function body evaluates in a context that holds the relevant indirect taints.

Example Consider the following function definition:

```
if (a) { f = function(){ var p = ...; return 0;};}
```

To ensure that taints from a flow into values returned by calls to f, the expression is rewritten to:

```
tmp = a,
TSET.indirect.push(tmp),
if (tmp) {
   tmp1 = TSET.indirect.get(),
   f = TSET.boxAndTaint(
         function(I){
           var TSET = new TSET();
           TSET.indirect.push(tmp1),
           TSET.indirect.push(I),
           p = TSET.boxAndTaint(...,...),
           TSET.boxAndTaint(box(0,...),...)
         },...)
},
TSET.indirect.pop()
```

where again, we elide some instrumentation for brevity. Again, the taints for the branch are pushed onto the indirect taint stack. Now, for the function, we create a new parameter for the indirect taints at the callsite, and a new local TSET object to which we add the indirect taints. The value returned by the function is boxed-and-tainted with all the taints at that point, which will include the indirect taints in tmp1 and hence, the guard a. Finally, note that the function object itself is boxed-and-tainted, before the taints from the guard get popped off.

Function Calls For each function call of the form $e(e_1 \ldots)$, we create temporaries to hold the values of the function and its arguments, and then call the function with the arguments and the current set of indirect taints (for the new indirect taint parameter I).

Example Consider the call expression: `if (b) { f() }` where f is bound to the function defined in the previous example. To ensure that taints from b flow into p (in the body of f) when the function is called, the expression is rewritten to

```
tmp = b,
TSET.indirect.push(tmp),
if (tmp) { f(TSET.indirect.get()) },
TSET.indirect.pop()
```

which has the effect of passing the indirect taints at the callsite to the (rewritten) callee, as the value of the parameter I which is added to the indirect taint stack as soon as the function begins execution, thereby flowing into boxed-and-tainted RHS expression which is then assigned to p.

4.5 Unboxed Objects

So far, we have assumed that we can always add a taint field to each object. Unfortunately, this assumption *only* holds for boxed values, and not for unboxed values. Indeed, in our implementation, we found that several DOM API functions are implemented in native code, and require unboxed objects as arguments. Thus, the rewriting must be capable of tracking taints for object-locations such as the cookie field of the document object, which is always unboxed (which precludes using document.cookie.taint).

To solve this problem, we observe that whenever the f field of an object must have an unboxed value, we can store the taints associated with that value in a special field taint_f of the "parent" object. Thus, for example, we store the taints associated with the cookie field of the document object inside document.taint_cookie.

Note that this scheme is robust to aliasing. This is because we only attach taints to the parent when the field f in question holds an unboxed object. As unboxed objects are immutable, we can safely assume that the f fields of two different parents refer to distinct copies of the unboxed object, and hence there is never any aliasing of unboxed fields.

Box-Sensitive Assignments: Fields To attach the taints with the "parent" object, we use a special box-sensitive assignment (implemented as a JavaScript library call, but shown as an infix operation for clarity). A *field box-sensitive assignment* $v_1[v_2] \leftarrow v_3$ assumes that v_3 is a boxed object. It checks if the v_1 object's v_2 field can be boxed. If so, the assignement is carried out as directly. If not *i.e.,* if the field must be unboxed, then v_3 is unboxed and the result assigned to the field and the taints associated with v_3 are assigned to the special field of the parent object v_1.taint_f. Notice that for field-reads $e_1[e_2]$, we add the taints from the special parent field (if any) to the top of the direct-taint stack, via the call

$$\texttt{TSET.direct.add}(\texttt{tmp}_1[\text{“}\texttt{taint\_}\text{”} + \texttt{tmp}_2], \texttt{url})$$

before returning the object corresponding to the field read.

With Scoping Box-sensitive assignments work well for field-assignments which contain an explicit reference to the host-object. The with-expressions of JavaScript allow programmers to access fields *implicitly*, without mentioning the host object. For example

$$\texttt{with}\ (\texttt{document})\ \{\texttt{location} = \texttt{x}\}$$

has the effect of assigning x to document.location, since the assignment occurs *in the scope of* the document object. Thus, even variable assignments must be box-sensitive, as they may be field-assignments within a with-expression, and to do so, we must track the object in whose scope an expression is evaluated.

With Stack (TSET.with) To track the object in whose scope any given assignment occurs, we augment the taint set object with a *with-scope* stack (named TSET.with). For with expressions of the form with (e_1) $\{e_2\}$, we push the object in whose scope the body is evaluated (namely the object e_1 evaluates to) onto the top of the stack (via a call to TSET.with.push()) and we pop it off the stack after the body has been evaluated, taking care to save and return the result via a temporary.

Box-Sensitive Assignments: Variables With this stack in place, we can handle *variable box-sensitive assignments* $v_1 \leftarrow v_2$ by treating them as field box-sensitive assignments $v[v_1] \leftarrow v_2$ where v is the object at the top of the current with stack, if v has a field named v_1. Otherwise, it is treated as a normal assignment.

Example The rewrite function converts the with-expression shown above (`with(document){...}`)) to:

```
tmp = document,
with(tmp){
    TSET.with.push(tmp),
    TSET.check(x, None, "location", ...),
    TSET.bsAsgn("location", x),
    TSET.with.pop()
  }
```

which is simplified for clarity. In the rewritten version the document object (via tmp) is pushed onto the with-stack. Next, a check is added to determine if the assignment is legal, and the actual assignment is converted into a box-sensitive assignment via an invocation of the bsAsgn method of TSET. Finally, the with-stack is reverted to its original state. The check and bsAsgn methods use the top of the with-stack to determine the target object-location. The former enforces the flow policy (and halts execution if x has the wrong taints), and the latter assigns causes the taints from x to flow into document.location.

5. Evaluation

We have implemented our dynamic rewrite-based information flow framework for JavaScript in the Chrome browser. 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 webpage 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.

5.1 Experimental setup

The 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.

Benchmarks We use the web sites from the latest Alexa top 100 list as our benchmark. Alexa is a web company which ranks web sites based on traffic. The web sites on the Alexa 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 top 100 list, and we visited many of them manually to make sure that they operate properly.

Policies 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 web site 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 websties, 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 web sites 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 web sites 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. 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.

5.2 Precision

Figure 6 shows the results of running our dynamic information flow framework on the Alexa 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.

Table format The columns in the table are as follows: "Site and rank" is the name of the web site and its rank in Alexa 100 list; "Total LOC" is the number of lines of JavaScript code on each website, including code from other sites, as formatted by our pretty printer built in Chrome; "Other Domain KLOC" is the number of lines of code from other sites; "# Taint Val" is the number of dynamically created tainted 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: "same" is for the same-origin policy; "dmn" is for the same-domain policy; "whlst" 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 we as 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. This is because all the JavaScript code for facebook comes from a web site fbcdn.net. This web site is not in the same domain as facebook, and it is only referenced by one web site 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 "same" to "dmn" to "'whlst', 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.

Comparison with previous results Previous work on information flow for JavaScript has shown how to detect policy violations in a hybrid static-dynamic approach called Staged Information Flow(SIF). We compare our purely dynamic results with the ones reported in the SIF work [3]. For the document.cookie location, there are some websites for which our tool detects flow, but SIF did not, and vice-versa. The SIF results show that there are several web sites in which the document.location policy is violated, whereas our dynamic approach has not uncovered any such violations. These differences can be explained by two factors. First, the SIF analysis had a static component, that was conservative in the face of various control-flow constructs, such as branches, loops, first-class functions and dynamic field accesses. Thus, the violations reported in [3] may well be false-positives. Our rewrite-based

Site and rank	Total KLOC	Other Domain KLOC			# Taint Val(k)			Cookie			Location		
		same	dmn	whlst	same	dmn	whlst	same	dmn	whlst	same	dmn	whlst
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	3.9	3.0	63.1	52.7	35.9	48	38	17	0	0	0

Figure 6. Precision results for a subset of the Alexa 100 websites. The last row summarizes results for all 100 sites.

framework, on the other hand, is purely dynamic and thus only reports real violations that actually occur. Second, there is a non-trivial amount of churn that happens on a regular basis on the web sites themselves and the JavaScript code that they run (not to mention that each visit to the web site, even seconds apart can return substantially different code because of different ads being introduced). Thus, the JavaScript code that our tool analyzed may be quite different from the code used in the SIF experiments.

5.3 Performance

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.

Timing Measurements 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 7 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* web sites are trusted.

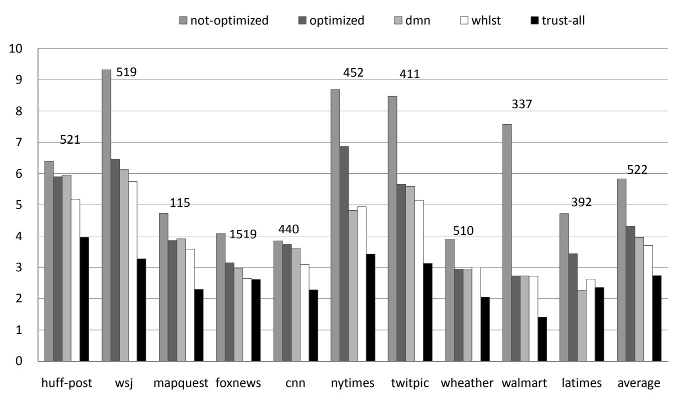
Optimizations We describe the three most important optimizations we performed for the "optimized" bar. First, in the `TSET.direct` stack, when there is a pop followed by a push, and just before the pop there are *no* taints stored at the top of the stack, we cache the object at the top of the stack before 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 savings. Second, 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.

JavaScript execution time The left chart of Figure 7 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. There are a few exceptions. For example, in the case of `huff-post` and `mapquest`, the "dmn" bar is slightly slower than the bar to its left. This is because for these benchmarks, the additional overhead that "dmn" introduces to check for sub-domains is not recouped by the savings from the drop in number of tainted objects.

The data from Figure 7 shows that our original `TSET` library slows down JavaScript execution significantly – anywhere from about 50% to just over 9X, and on average about 6X. The optimized `TSET` library provides significant performance gains over the original library. 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 web site 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 7 shows the *total* execution time of 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 7 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 7 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 2.4 times slower.

Conclusions Our experiments show that dynamic information flow control is feasible, but imposes a perceptible overhead in the running time. However, simple optimizations have yielded significant improvements in the run-time, and much of the overhead is masked by the latency of downloading pages. Thus, we believe that with



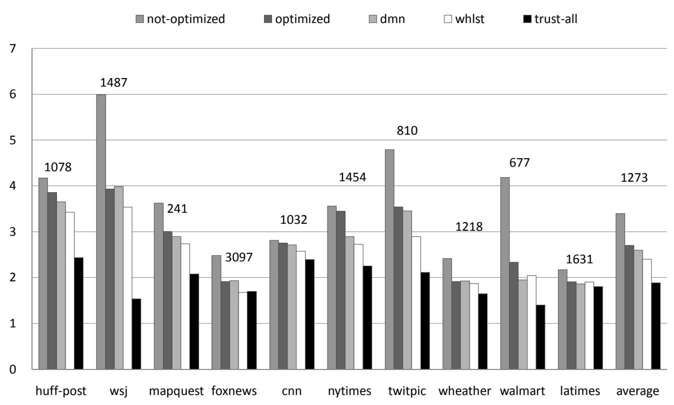


Figure 7. Slowdown for JavaScript (left) and Page Loading (right)

careful engineering, dynamic rewriting based policy checking can become a practical way to secure JavaScript web-applications.

6. Related Work

Information flow [4] and non-interference [5] 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 [19, 13], model checking [16], Hoare-logics [1], and dataflow analyses [8, 14]. Of these, the most expressive policies are captured by the dependent type system of [11], 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 [15, 17], virtual machines [2], and binary rewriting [12]. [18] modifies the browser's JavaScript engine to track a taint bit that determines whether a piece of data is sensitive, and reports an XSS attack if this data is sent to a domain other than the page's domain. Our approach provides a different point in the design space than [18]. 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. Our approach is also implemented using a JavaScript rewriting strategy rather than modifying the JavaScript run-time. Because of all of these reasons, our approach also has a larger performance overhead.

Many web application exploits are caused by not correctly sanitizing user generated content on the *server*. Several authors have proposed static analyses that determine if user generated content has been correctly sanitized [20]. Gatekeeper [9] is a static analysis that finds bugs on the JavaScript code known on the server.

One way to ensure safety on the *client* is to disallow unknown scripts from execution [6]. However, this will likely make it hard to use dynamic third-party content. Finally [21] 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 [7]. These policies are quite different from information flow in that they require sparser instrumentation, and cannot enforce fine-grained isolation.

References

[1] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *SAS*, pages 100–115, 2004.

[2] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, pages 50–62, 2009.

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

[5] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[6] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[7] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *APLAS 08*, pages 326–341, 2008.

[8] M. S. Lam, M. Martin, V. B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *PEPM*, pages 3–12, 2008.

[9] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. Technical Report MSR-TR-2009-16, Microsoft Research, Feb. 2009.

[10] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *APLAS*, pages 307–325, 2008.

[11] A. C. Myers. Programming with explicit security policies. In *ESOP*, pages 1–4, 2005.

[12] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[13] F. Pottier and V. Simonet. Information flow inference for ml. In *POPL*, pages 319–330, 2002.

[14] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.

[15] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[16] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, pages 352–367, 2005.

[17] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, 2004.

[18] 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 *NDSS*, 2007.

[19] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *POPL*, 2000.

[20] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[21] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL*, pages 237–249, 2007.