JavaScript Instrumentation in Practice

Haruka Kikuchi   Dachuan Yu   Ajay Chander   Hiroshi Inamura   Igor Serikov
DOCOMO Communications Laboratories USA, Inc.
3240 Hillview Avenue, Palo Alto, CA 94304

DOCOMO USA Labs Technical Report DCL-TR-2008-0053

Abstract

JavaScript provides useful client-side computation facilities, enabling richer and more dynamic web applications. Unfortunately, the power and ubiquity of JavaScript has also been exploited to launch various browser-based attacks. Our previous work proposed a theoretical framework applying policy-based code instrumentation to JavaScript. This paper further reports our experience carrying out the theory in practice. Specifically, we discuss how the instrumentation is performed on various JavaScript and HTML syntactic constructs, present a new policy construction method for facilitating the creation and compilation of security policies, and document various practical difficulties arose during our prototyping. Our prototype currently works with several different web browsers, including Safari Mobile running on iPhones. We report our results based on experiments using representative real-world web applications. Although discussing a particular prototype, we believe the techniques therein will also be useful to other studies on JavaScript security.

1 Introduction

Web browsing is becoming an indispensable part of our daily lives. Besides providing instant access to various information, the Web is also evolving into a major software and service platform. Users access the platform, and sometimes participate in service building, through web browsers running on client machines, including some mobile devices.

The success of the Web can be contributed partly to the use of client-side scripting languages, such as JavaScript [3]. Programs in JavaScript are deployed in HTML documents. They are interpreted by web browsers on the client machine, helping to make web pages richer, more dynamic, and more "intelligent."

As a form of mobile code, JavaScript programs are often provided by parties not trusted by web users, and their execution on the client systems raises serious security concerns. The extent of the problem is well known [2, 16], ranging from relatively benign annoyances (*e.g.,* popping up advertising windows, altering browser configurations) to criminally serious attacks (*e.g.,* XSS [13], phishing [15]).

In previous work [21], we formally studied the application of program instrumentation to enforcing security policies on JavaScript programs. Specifically, we clarified the execution model of JavaScript (particularly, higher-order script—script that generates other script at runtime), presented its instrumentation as a set of syntactic rewriting rules, and applied edit automata [12] as a framework of policy management. Focusing on articulating the generic instrumentation techniques and proving their correctness, the previous work is necessarily abstract on several implementation aspects, and can be realized differently when facing different tradeoffs. For example, among other possibilities, the instrumentation process can be carried out either by a browser on the client machine or by a proxy sitting on the network gateway.

In this paper, we discuss the practical application of the above theory to real-world scenarios. Specifically, we have completed a relatively mature prototype following a proxy-based architecture. In this prototype, both the instrumentation process and the policy input are managed by a proxy program, which is situated on the network gateway (or an enterprise firewall) and is maintained separately from the JavaScript programs of concern. The browser is set up to consult the proxy for all incoming traffic, and the proxy sends instrumented JavaScript programs (and HTML documents) to the browser.

Such an architecture provides centralized interposition and policy management through the proxy, thus enabling transparent policy creation and update for the end user. It requires no change to the browser implementation; therefore, it is naturally applicable to different browsers, modulo a few browser-specific implementation issues. As part of our experiments, we applied our proxy to the Safari Mobile browser on an iPhone [1]. The instrumentation and security protection worked painlessly, even though no software or browser plug-in can be installed on an iPhone. Furthermore, the proxy-based architecture poses minimal computation requirement on the client device for securely rendering web pages (as shown in our experiments, some popular web pages contain several hundred kilobytes of JavaScript code; their instrumentation takes a nontrivial amount of time). It is thus suitable for deployment with the use of mobile browsers.

There has been other work (*e.g.,* BrowserShield [17], AjaxScope [9]) on JavaScript instrumentation. Although similar in nature to our work, they are motivated differently and, as a result, differ in the overall design. BrowserShield provides automatic vulnerability filtering on a firewall. Part of the rewriting and interposition logic is carried out on the client machine with help of a browser plug-in. In addition, its policies are directly written in the form of JavaScript functions, and mainly target security experts who develop vulnerability signatures and patches. AjaxScope, on the other hand, provides debugging help to JavaScript program developers. It thus targets non-malicious code, and does not address the handling of dynamically generated code (higher-order script). In contrast, our work uniquely targets user policies and their enforcement on mobile devices, across different browsers, without the installation of any client-side software or plug-ins. Two of our major differentiating factors include usable policy management (we use a policy compiler to compile browser-based security policies into rewriting rules) and proxy-centric runtime instrumentation (leveraging the proxy's computation power, we trade on-device computation with on-demand communication with the proxy).

We have successfully run our prototype with several browsers: Firefox, Konqueror, Opera, Safari, and Safari Mobile on an iPhone [1]. IE is currently only partially supported, because its behaviors on certain JavaScript code are significantly different than those of the other tested browsers (details will be discussed in Section 5.4). For the experiments, we applied a selected set of policies, which triggered the instrumentation of a variety of JavaScript constructs. We hand-picked some representative real-world web pages as the instrumentation target. Measurements are made on both the proxy overhead and the client overhead. The initial numbers matched our expectations, noting that our prototype had not been optimized for performance.

The remainder of this paper is organized as follows. Section 2 reviews the formal theory of JavaScript instrumentation, and clarifies some key aspects of the realization effort to be expanded upon in the next three sections. Section 3 presents the application of the theoretical instrumentation method to the actual JavaScript language. Section 4 details how we use some domain-specific abstractions for policy construction. Section 5 describes a few interesting implementation issues raised by the proxy-centric requirements. Experiments are discussed in Section 6. We compare with related work in Section 7, and finally conclude in Section 8.

2 Background

2.1 Previously: a Theoretical Framework

The generic theoretical framework for JavaScript instrumentation is illustrated in Figure 1 (reused from previous work [21]). Beyond the regular JavaScript interpreter provided in a browser, three modules are introduced to carry out the instrumentation and policy enforcement. A rewriting module ι serves as a proxy between the browser and the network traffic. Any incoming HTML document D, possibly with JavaScript code embedded, must go through the rewriting module first before reaching the browser.

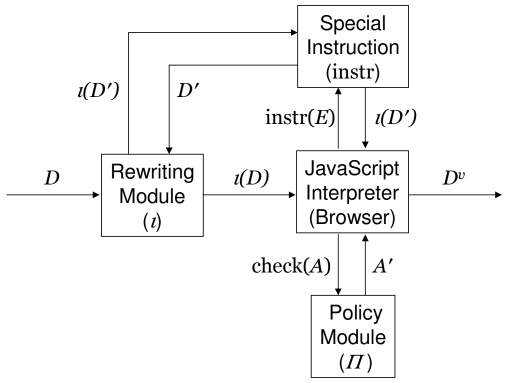


Figure 1: JavaScript instrumentation for browser security

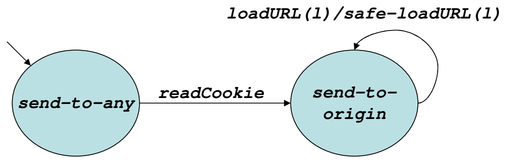


Figure 2: Example edit automaton for a cookie policy

The rewriting module identifies security-relevant actions A out of the document D and produces instrumented code $check(A)$ that monitors and confines the execution of A. This part is not fundamentally different from conventional instrumentation techniques [20, 4]. However, this alone is not sufficient for securing JavaScript code, because of the existence of higher-order script (*e.g.,* $document.write(E)$, which evaluates E and use the result as part of the HTML document to be rendered by the browser)—some code fragments (*e.g.,* those embedded in the value of E) may not be available for inspection and instrumentation until at runtime. To address this, the rewriting module wraps higher-order script (*e.g.,* E) inside a special instruction (*e.g.,* $instr(E)$), which essentially marks it to be instrumented on demand at runtime.

With the rewriting module as a proxy, the browser receives an instrumented document $\iota(D)$ for rendering. The rendering proceeds normally using a JavaScript interpreter, until either of the two special calls inserted by the rewriting module is encountered. Upon $check(A)$, the browser consults the policy module Π, which is responsible for maintaining some internal state used for security monitoring, and for providing a replacement action A' (a secured version of A) for execution. Upon $instr(E)$, the implementation of the special instruction $instr$ will evaluate E to D' and invoke the rewriting module at runtime to perform the necessary rewriting on higher-order script. The instrumented $\iota(D')$ will then be sent back to the browser for rendering, possibly with further invocations of the policy module and special instruction as needed.

The policy module Π essentially manages an edit automaton [12] at runtime for security enforcement. A simple policy is illustrated in Figure 2, which restricts URL loading to prevent potential information leak after cookie is read. Following this, the policy module updates the automaton state based on the input actions A that it receives through the policy interface $check(A)$, and produces output actions A' based on the replacement actions suggested by the automaton. For example, if the current state is `send-to-origin`, and the input action is `loadURL(l)`, then the current state remains unchanged, and the output action becomes `safe-loadURL(l)`, which performs necessary security checks and user promptings before loading a web page. Interested readers are referred to the previous work [21] for why edit automata serve as a suitable policy framework, what their capabilities are, and how to support their combination and update.

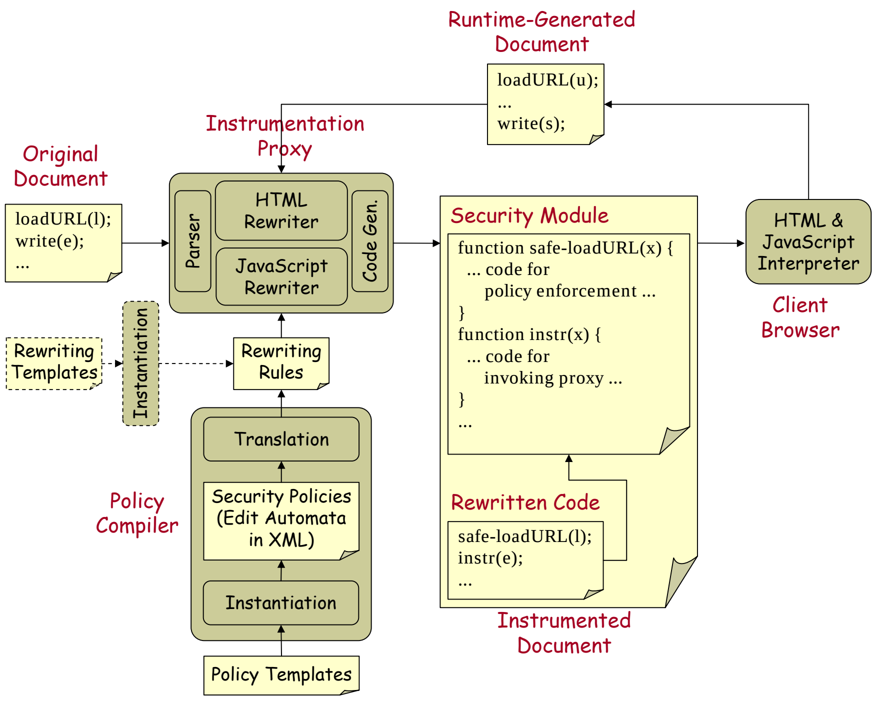


Figure 3: A proxy-centric realization

2.2 This Paper: a Proxy-Centric Realization

Focusing on the formal aspects and correctness of policy-based instrumentation on higher-order script, the previous work is largely abstract on the implementation aspects. Subsequently, we have conducted thorough prototyping and experiments on the practical realization of the formal theory. Several interesting topics were identified during the process, which we believe will serve as useful contributions to related studies.

Overall, we have opted for a proxy-centric realization for its browser independence and low computation overhead on client devices, as has been discussed in Section 1. This can be viewed as an instantiation of the framework in Figure 1 for a specific usage scenario—the use with multiple mobile browsers. The instantiated architecture is more accurately depicted in Figure 3. The basic requirement here is that the proxy takes care of the rewriting and policy input, without changing the browser implementation. As a result, the tasks of the policy module Π and the special instruction $instr$ in Figure 1 have to be carried out in part by regular JavaScript code (the Security Module in Figure 3). Such JavaScript code is inserted into the HTML document by the proxy based on some policy input.

The rewriting process is carried out on the proxy using a parser, two rewriters, and a code generator. The rewriters work by manipulating abstract syntax trees (ASTs) produced by the parser. Transformed ASTs are converted into HTML and JavaScript by the code generator before fed to the browser that originally requested the web pages. The actual proxy contains also a processor for handling outgoing requests from the client browser, which is largely straightforward, thus omitted from the figure.

The rewriting rules direct the rewriters to put in different code for the security module when addressing different policies. On the one hand, the rewriters need to know what syntactic constructs to look for in the code and how to rewrite them. Therefore, they require low-level policies to work with syntactic patterns. On the other hand, human policy designers should think in terms of high-level actions and edit automata, not low-level syntactic details. We use a policy compiler to bridge this gap, which is a stand-alone program that can be implemented in any language. Note that policy compilation happens statically (off-line).

We now summarize three major technical aspects of this proxy-centric realization. These will be expanded upon in the next three sections, respectively.

The first and foremost challenge is raised by the flexibility of the JavaScript language. Much effort is devoted to identifying security-relevant actions out of JavaScript code—there are many different ways of carrying out a certain action, hence many different syntactic constructs to inspect. We introduce a notion of *rewriting templates* to help managing this at a somewhat abstract level. Policy designers instantiate rewriting templates to rewriting rules, which in turn guides the instrumentation. This allows us to stick to the basic design without being distracted by syntactic details. A similar aspect is on the identification of JavaScript code out of HTML documents, as well as higher-order script out of JavaScript code. These can both be handled uniformly using rewriting templates.

Next, the rewriting templates can be viewed as an interface for policy writing, because they essentially specify rewriting rules and replacement actions. However, they are still too low-level to manage in practice. In contrast, the theoretical framework used edit automata for policy management. We support edit automata using an XML representation, and compile them into rewriting rules. As a result, we can handle all policies allowed by the theoretical framework. Nonetheless, edit automata are general-purpose, and browser-based security policies could benefit further from domain-specific abstractions. We have identified some useful patterns of policies during our experiments. We organize these patterns as *policy templates*, which are essentially templates of specialized edit automata. We again use an XML representation for policy templates, which enables a natural composition of simple templates to form compound ones.

Finally, using a proxy, we avoid changing the browser implementation. This provides platform independence and reduces the computation overhead on the client devices. As a tradeoff, there are some other difficulties. In particular, the interfaces to the special instruction $instr$ and the policy module Π have to be implemented in regular JavaScript code and inserted into the HTML document during instrumentation. Such code must be able to call the proxy at runtime, interact with the user in friendly and meaningful ways, and be protected from malicious modifications (it resides in the same environment as incoming, potentially malicious JavaScript code). Furthermore, different browsers sometimes behave differently on the same JavaScript code, an incompatibility that we have to address carefully during instrumentation.

3 Rewriting JavaScript and HTML

We now describe the details of the instrumentation. Recall that the instrumentation is carried out by transforming abstract syntax trees (ASTs). Specifically, the proxy identifies pieces of the AST to be rewritten, and replaces them with new ones that enforce the desired security policies. Both action replacement (from A to $check(A)$) and higher-order script handling (from E to $instr(E)$) are addressed during the process.

3.1 Action Replacement via Rewriting Templates

Although a clean process in Figure 1, action replacement in the actual JavaScript language raises interesting issues due to the flexible and dynamic nature of the language. Specifically, actions A in Figure 1 exhibit themselves in JavaScript and HTML through various syntactic forms. Some common examples include property access, method calls, and event handlers. Therefore, upon different syntactic categories, the instrumentation should produce different target code, as opposed to the uniform $check(A)$. We specify what syntactic constructs to rewrite and how to rewrite them using *rewriting rules* (see Figure 3).

We summarize some commonly used rewriting rules for various syntactic constructs in Table 1 as *rewriting templates*. The first column of the table shows the template names and parameters. The second shows the corresponding syntactic forms of the code pieces to be rewritten (which correspond to actions A). The third shows the target code pieces used to replace the original ones (which correspond to $check(A)$). These

Table 1: Rewriting templates

Rewriting templates	**Sample Code patterns**	**Rewritten code (redirectors)**	**Sample instantiation**
(Get, $obj, prop$)	$obj.prop$ $obj[$“$prop$”$]$	sec.GetProp($obj, prop$)	(Get, document, cookie) (Get, window, alert)
(Call, $obj, meth$)	$obj.meth(E, \dots)$ $obj[$“$meth$”$](E, \dots)$	sec.CallMeth($obj, meth, E_i, \dots$)	(Call, window, open)
(GetD, $dprop$)	$dprop$	sec.GetDProp($dprop$)	(GetD, location)

(CallD, $dmeth$)	$dmeth(E, \dots)$	sec.isEval($dmeth$) ?
		eval(sec.instrument($E_i, \dots$)) :
		sec.CallDMeth($dmeth, E_i, \dots$)
			(CallD, open)

(Set, $obj, prop$)	$obj.prop = E$
	$obj[$“$prop$”$] = E$
	$obj.prop \mathrel{+}= E$
	$obj[$“$prop$”$] \mathrel{+}= E$
		sec.SetProp($obj, prop, E_i$)
		
		sec.SetPropPlus($obj, prop, E_i$)
			(Set, document, cookie)
			(Set, window, alert)

(SetD, $dprop$)	$dprop = E$
	$dprop \mathrel{+}= E$
		sec.SetDProp($dprop, E_i$)
		sec.SetDPropPlus($dprop, E_i$)
			(SetD, location)
			(SetD, open)

(Event, $tag, attr$)	<tag $attr$=“E”>	<tag $attr$=“sec.Event(this, E_i)”>	(Event, button, onclick)

(FSrc)	<img src=“U”
	onerror=“E”>
	<iframe src=“U”
	onload=“E”>
		<img src=“ ” onerror=“setAttribute(onerror, “E_i”);
		sec.FSimg(this, U)”>
		<iframe src=“ ” onload=“setAttribute(onload, “E_i”);
		sec.FSiframe(this, U)”>	

templates are to be instantiated to rewriting rules using relevant JavaScript entities and code. Some examples are given in the last column. Given instantiated rewriting rules, the rewriters will identify patterns of the second column, and produce instrumented code according to the third column.

For example, the first row (Get, $obj, prop$) specifies how to rewrite syntactic constructs of reading properties (both fields and methods). Get is the name of the template, and obj and $prop$ are parameters to be instantiated with the actual object and property. Two sample instantiations are given. The first is on the field access document.cookie. This is useful if one wishes to monitor cookie access for identifying cookie stealing behaviors. Based on (Get, document, cookie), the JavaScript rewriter will look for all AST pieces of property access (*i.e.,* those of the shape $obj.prop$ and $obj[$“$prop$”$]$), and replace them with a call to a *redirector* function sec.GetProp. Here sec is a JavaScript object inserted by our proxy; it corresponds to the “Security Module” in Figure 3. Among other tasks, sec maintains a list of private references to relevant JavaScript entities, such as document.cookie. The body of the redirector above will inspect the parameters obj and $prop$ at runtime to see if they represent document.cookie (if obj and $prop$ are statically known to be document and cookie, respectively, the inspection code is omitted for efficiency). If yes, the redirector proceeds to carry out a replacement action supplied during template instantiation.

The implementation of the replacement action is the topic of Section 4. For now, it suffices to understand the replacement action simply as JavaScript code. It can perform computation and analysis on the arguments of the redirector, provide helpful promptings to the user, and/or carry out other relevant tasks. One typical task that it carries out is to advance the monitoring state of the edit automaton used by the security policy.

The second example of the Get category is on accessing window.alert. Note that JavaScript allows a method to be accessed in the same way as a field. For example, var f = window.alert assigns the method window.alert to a variable f. This is handled during rewriting using the same Get category as described above, and the body of sec.GetProp can monitor such access and implement related policies (*e.g.,* to replace the access to window.alert with the access to an instrumented version sec.alert).

The remainder of the table follows the same intuition. (Call, $obj, meth$) tells the JavaScript rewriter to look for syntactic categories relevant to method calls (*e.g.,* $obj.prop(E, \dots)$, $obj[$”$prop$”$](E, \dots)$) in the AST and produce a call to a redirector `sec.CallMeth`. The argument E to the method invocation is rewritten to E_i following the same set of rewriting rules (the same also applies to most other cases in the table). For the sample instantiation $(\texttt{Call}, \texttt{window}, \texttt{open})$, as is the case of the field access example earlier, the redirector inspects the parameters obj and $prop$ at runtime to see if they match the template parameters `window` and `open`, and carries out the replacement action specified by the security policy.

$(\texttt{GetD}, \texttt{dprop})$ and $(\texttt{CallD}, \texttt{dmeth})$ are for accessing default properties and calling default methods. They are similar to the `Get` and `Call` templates, except they require only one parameter. $(\texttt{Set}, \texttt{obj}, \texttt{prop})$ and $(\texttt{SetD}, \texttt{dprop})$ are for setting object properties and default properties. Although it is possible to treat $+=$ and other similar constructs as syntactic sugars and expand them carefully (think side-effects) during the parsing phase, we chose to directly handle them in the rewriting rules for efficiency.

Sometimes relevant actions are coded as event handlers of certain HTML tags. For example, the code $<$`button onclick` $=$ "`alert()`"$>$ raises an alert window whenever the button is clicked. The template $(\texttt{Event}, tag, attr)$ captures such event handling constructs. The redirector `sec.Event` takes the instrumented expression E_i and the parent object `this` of the attribute as arguments. The exact implementation of the redirector depends on the security policy. Typically, the internal state of the policy edit automaton is updated while entering and exiting the corresponding event.

The last template $(\texttt{FSrc})$ is on the loading of external resources, such as those initiated by $<$`img src`$=$"E"$>$ and $<$`iframe src`$=$"E"$>$. Instead of directly loading such an external resource, we use an event handler with a redirector for interposition. The `src` attribute is modified from the original URL U to a space character to trigger immediately a corresponding event (*e.g.,* `onerror` for `img`, `onload` for `iframe`). In the rewritten event handler, after binding the original event handler (rewritten from E to E_i) back to the event for execution later, we call a redirector function (*e.g.,* `sec.FSimg`, `sec.FSiframe`). Although this $(\texttt{FSrc})$ template takes no parameter, one must still provide proper code to implement the redirector. For example, the redirector may check the target domain name of the URL U to identify where the HTTP request is sent, perform other URL filtering, and/or present user prompting. In general, this rule is also applicable to some other cases of HTTP requests (*e.g.,* $<$`a href` $=$ "U"$>$), except different event handlers (*e.g.,* `onclick`) and redirectors (*e.g.,* `sec.FShref`) are used accordingly.

3.2 Higher-Order Script and Built-in Rules

Although designed mainly for rewriting actions (A), the rewriting templates are also applicable for handling higher-order script. For example, `document.write`(E) should be rewritten using `sec.instrument`[1] (the realization of the `instr` of Figure 1). This can be represented using the `Call` template as $(\texttt{Call}, \texttt{document}, \texttt{write})$. Some code is provided in companion as the body of the corresponding redirector; that code feeds E_i (statically rewritten from E) into the rewriting interface `sec.instrument` (to be sent to the proxy and rewritten at runtime). Similar observation applies to some other forms of higher-order script, such as those through `eval` and `innerHTML`. In our prototype, however, the handling of higher-order script is directly coded in the rewriter for efficiency, since it is always performed regardless of what the security policy is.

There are some other cases where built-in rewriting rules are applied. Selected ones are given in Table 2. Most of the cases are designed to handle script that is not available statically, but rather generated or loaded at runtime. Others are for facilitating error handling and instrumentation transparency.

$(\texttt{HOScript})$ captures common cases of higher-order script as discussed above. Note that our parser pieces together consecutive `document.write` statements by concatenation their string arguments.

$(\texttt{NewObj})$ is used because JavaScript code may be introduced through function object creation. For example, `f` $=$ `new Function`("*JavaScript code goes here*") creates an object `f` as an executable function. Naturally, the code introduced as the string argument must also be instrumented. Following rule $(\texttt{NewObj})$,

[1]The prototype uses two functions to instrument HTML and JavaScript. We refer to them uniformly as `sec.instrument`.

Table 2: Selected built-in rewriting rules

Rewriting rules	**Sample Code patterns**	**Rewritten code**
(`HOScript`)	`document.write`(E) `eval`(E)	`eval(sec.instrument(`E_i`))`
(`NewObj`)	`new` $obj(E, \dots)$	`sec.NewObj`$(obj, E_i, \dots)$
(`JSURL`)	`<a href="javascript :` E`">`	`<a href="javascript :` E_i`">`
(`ExtScript`)	`<script src="`$d.p$`"></script>`	`<script src="`$d.p_i$`"></script>`

(`Chunk`)	`<script>`E`</script>`	`<script>`
		`ChunkBegin();` E_i`; ChunkEnd()`
		`</script>`

(`ForIn`)	`for` $(prop$ `in` $obj)$ $\{E\}$	`for` $(prop$ `in sec.ForIn(`obj`))` $\{E_i\}$
(`With`)	`with` (obj) $\{E\}$	`eval(sec.instrument(`obj, E_i`))`

a redirector `sec.NewObj` is applied to perform function object creation without by-passing necessary rewriting. In particular, `sec.instrument` will be called at appropriate places for higher-order script.

JavaScript code may also appear where a URL is expected (a JavaScript pseudo-protocol). For instance, `<a href="javascript :` E`">`, when activated, executes the code E. This is handled by (`JSURL`).

(`ExtScript`) handles the inclusion of external JavaScript files through `<script src="`$d.p$`"></script>`, where d is a domain (*e.g.,* `usenix.org`) and p is a path to a JavaScript file (*e.g.,* `xyz.js`). We rewrite the access to target a corresponding file on the proxy, which is an instrumented version of the original source.

(`Chunk`) identifies script chunks from an HTML document, and inserts prologues and epilogues. This is to prevent malicious ungrammatical script from circumventing the instrumentation, such as in the following:

`<script>document.write("<script>`*malicious code goes here*`</scr");<script>`$\underline{\texttt{ipt>}}$

Since the argument of `document.write()` is not a well-formed piece of JavaScript code, our proxy will treat it as a string and thus not instrument it. Nonetheless, `chunkEnd()` will insert a special character at the end of the script chunk. As a result, the string argument of `document.write()` will not be pieced together with the underlined part when rendered by the browser, thus avoiding the execution of the malicious code.

In the proxy-based architecture, we have to load the redirectors (the embodiment of the security policies) into the same execution environment as the instrumented document. All the redirectors are organized in the `sec` module, which is attached to incoming document as a property of the `window` object. The `for`...`in` construct provides access to all properties of an object. Therefore, incoming JavaScript may use it to access the `sec` module, either maliciously or unintentionally. The `ForIn` template rewrites `for`...`in` constructs to avoid such access. Specifically, the redirector `ForIn` returns a list of properties that does not contain `sec`.

The final rule (`With`) is for `with` statements. It is tempting to treat `with` as pure syntactic sugars, for example, expanding `with (a) {x = 1;}` to `a.x = 1;` and perform the rewriting therefrom. Unfortunately, this may fail to produce the right behavior in the presence of higher-order script, such as in the following:

`with (a) { eval("x = 1;") }`

A variable `x` appears in the argument of `eval` as a string. In general, the string may not be available for analysis until at runtime. Therefore, such a `with` statement cannot be fully expanded statically. In essence, `with` adds its argument `obj` onto the current scope chain, and evaluates its body `E` using the new scope chain. Therefore, when the (`With`) rule passes the argument E_i (statically rewritten from E) to `sec.instrument` for runtime instrumentation, an additional parameter obj is provided. As a result, `sec.instrument` appends obj onto the current scope chain before feeding E_i to the proxy. Naturally, the scope chain is an important piece of input to the proxy. We have elided this detail so far for ease of understanding, since no other rewriting case requires any nontrivial handling of the scope chain.

In summary, the proxy performs rewriting by syntax-directed pattern matching on ASTs, and redirectors are used to implement appropriate interposition logic and security polices. Built-in rewriting rules are always applied, but a policy designer may customize action replacement using rewriting templates. If the same rewriting template is instantiated multiple times on different entities, a single rewriting rule with a merged function body for the redirector is produced. The proxy only rewrites cases described by the given rewriting rules. If a certain security policy only uses one rewriting template (*e.g.,* `get`, possibly with multiple instantiations), then only one rewriting rule is produced, and only the corresponding syntactic constructs (*e.g.,* `obj.prop`, `obj["prop"]`) are rewritten. This avoids unnecessary rewriting and execution overhead.

4 Policy Writing and Management

The rewriting templates and their redirector code in Section 3.1 serve as a low-level policy mechanism. It allows policy designers to focus on the abstract notion of "actions" without being distracted by the idiosyncrasies of JavaScript syntax. However, it provides little help on encoding edit automata. If used for construction, a policy designer needs to implement states and transitions of policy automata in the redirectors.

We now discuss a more manageable policy framework that directly accommodates the notion of edit automata, allowing policy designers to focus more on implementing replacement actions (*e.g.,* insertion of runtime checks). The key of this framework is the policy compiler in Figure 3, a stand-alone program that compiles high-level policies into rewriting rules off-line.

4.1 Edit Automata

In essence, an edit automaton concerns a set of states (with one as an initial state), a set of actions, and a set of state transitions. In any state, an input action determines which transition to take and which output action to produce. We use an XML file to describe all these aspects for an edit automaton as a security policy.

For example, the edit automaton of Figure 2 is represented in XML as shown in Figure 4. This edit automaton consists of an initial state and some transitions. Each transition describes a single edge in Figure 2. The first transition has three components: an (input) action, a from-state, and a to-state. Upon an input action that matches the rewriting template (`Get`, `document`, `cookie`) described by the action, if the current state is `s1`, then the edit automaton goes to state `s2`. Since the particular edit automaton specifies no replacement action for this transition, the original action will be carried out. In general, however, a replacement action may also be specified, as is the case of the second transition. Note that the code for the replacement action may refer to the formal parameters of the corresponding redirector as specified in Table 1.

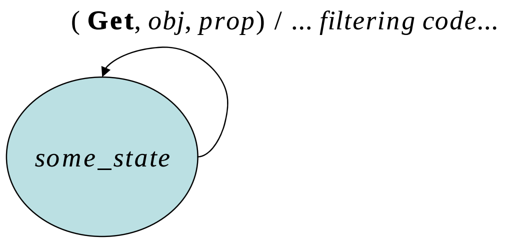
The compilation from a such represented edit automaton to rewriting rules goes as follows. The input action is identified by the template name and parameters. Based on Table 1, there will be a corresponding rewriting template to instantiate and a redirector to supply code as the body. The code performs state transition based on the from/to states, and invokes the replacement action. More specifically, for the above discussed transition, the policy compiler would produce the rewriting rule (`Get`, `document`, `cookie`) and insert into the redirector `sec.GetProp` some code of the following shape:

```
if (sec.compare(obj.prop, "document.cookie")) {
    if (sec.state == s1)   sec.state = s2;
}
return obj.prop;
```

Here obj and $prop$ stands for the formal parameters of `sec.GetProp`. `sec.compare` is a utility function called at runtime to decide the identity of $obj.prop$. The string "`document.cookie`" is used for clarity here,

```
<EA>
  <initial> s1 </initial>
  <transition>
    <action>
      <category> Get </category>
      <object> document </object>
      <property> cookie </property>
    </action>
    <from> s1 </from>
    <to> s2 </to>
  </transition>
  <transition>
    <action>
      <category> fsrc </category>
    </action>
    <from> s2 </from>
    <to> s2 </to>
    <replacement>
      ...code of safeloadURL...
    </replacement>
  </transition>
</EA>
```

Figure 4: An edit automaton in XML



```
<template name = "ReadAccessFiltering">
  <object> obj </object>
  <property> prop </property>
  <states> some_state </states>
  <replacement>
    ...filtering code...
  </replacement>
</template>
```

Figure 5: Read access filtering

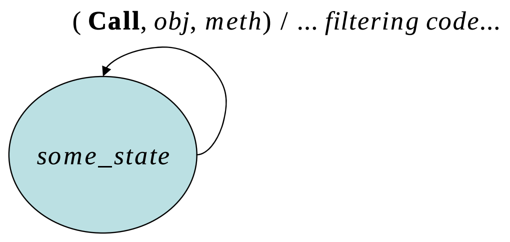
but the actual implementation works directly with a private reference to `document.cookie`. `sec.state` is the state of the edit automaton maintained by the security module, and `s1` and `s2` are state constants. The state transition from `s1` to `s2` applies if *obj.prop* is indeed `document.cookie`. The original action *obj.prop* is carried out unconditionally, because the edit automaton specifies no replacement action for this case.

After the edit automaton is compiled into rewriting rules and redirectors, the rewriters described in Section 3 takes cover, rewriting all relevant syntactic constructs using the corresponding redirectors.

4.2 Policy Templates

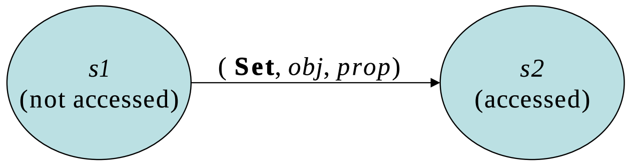
During our policy experiments, we identified several commonly used patterns of edit automata. We organize these as *policy templates*. Instead of always describing an edit automaton from scratch, a policy designer may instantiate a relevant template to quickly obtain a useful policy. We illustrate this with examples.

One pattern is on filtering read access to object properties (both fields and methods). The (fragment of) edit automaton is shown in Figure 5, together with an XML representation. Note that this is essentially a template to be instantiated using a specific object, property, state, and filtering code. It captures a common pattern where the property access is filtered without applying any state transition.



```
<template name = “CallFiltering”>
  <object> obj </object>
  <property> meth </property>
  <states> some_state </states>
  <replacement>
    ... filtering code ...
  </replacement>
</template>
```

Figure 6: Call filtering



```
<template name = “WriteAccessTracking”>
  <object> obj </object>
  <property> prop </property>
  <states> s1, s2 </states>
</template>
```

Figure 7: Write access tracking

One may also filter method calls in a similar pattern (Figure 6). This is typically used to insert security checks for method calls. Again, the template is to be instantiated with specific entities.

Another pattern is on tracking write access to properties (Figure 7). This tracks the execution of property writing by transitioning the state of the automaton. The replacement action is the same as the input action. The corresponding policy template specifies two states, but no replacement action.

We have implemented a total of 14 policy templates, including four on tracking property access (object and default properties; read and write access), four on filtering property access (object and default properties; read and write access), four on method calls (tracking and filtering; object and default methods), one on tracking inlined event handlers (e.g., `<body onunload=“window.open();”>`), and one on filtering implicit HTTP requests (e.g., images and inlined frames). These policy templates proved useful during our experiments, but new ones can always be constructed. The policy compiler expands instantiated policy templates into edit automata, and further compilation follows as introduced earlier.

4.3 Compound Templates

The above templates each describe a particular aspect of security issue. It is often the case that multiple templates are used together to enforce a useful policy. As an example, consider a simple policy of adding a prefix “`Security Module :` ” to the alert text produced by `window.alert`. Naturally, we implement a replacement action `myAlert` as follows: `function myAlert(s) { window.alert(“Security Module : ” + s); }`.

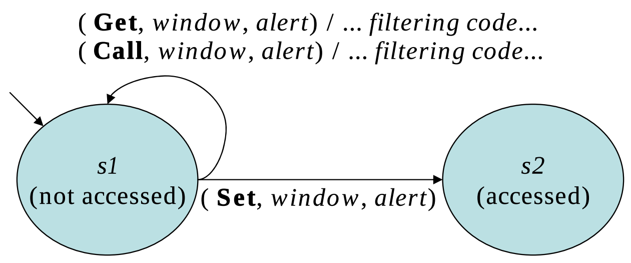
This obviously requires some filtering on read access and calls to `window.alert`, as illustrated in Figures 5 and 6. For example, the former is applicable to rewriting code from `f = window.alert` to `f = myAlert`, and the latter is applicable to rewriting code from `window.alert(“Hello”)` to `myAlert(“Hello”)`.

An additional complication is that JavaScript code in the incoming document may choose to rewrite `window.alert` for other functionalities:

```
window.alert = function(s) {};
window.alert(”a debugging message to be ignored”);
```

Here, `window.alert` is redefined as an “ignore” function. It would be undesirable to perform the same filtering after the redefinition. Therefore, a more practical policy is to filter read access and calls to `window.alert` only if it has not been redefined.[2] This can be addressed using write access tracking as in Figure 7.

[2]Readers might wonder if this could be exploited to circumvent the instrumentation. The answer is no, because the function


(**Get**, *window*, *alert*) / ... *filtering code...*
(**Call**, *window*, *alert*) / ... *filtering code...*
s1
(not accessed)
(**Set**, *window*, *alert*)
s2
(accessed)

```
<policy>
  <initial> s1 </initial>
  <template name = “ReadAccessFiltering”>
    <object> window </object>
    <property> alert </property>
    <states> s1 </states>
    <replacementAction>
      ... filtering code returns myAlert ...
    < /replacementAction>
  </template>
  <template name =″ CallFiltering”>
    <object> window </object>
    <property> alert </property>
    <states> s1 </states>
    <replacementAction>
      ... filtering code calls myAlert ...
    < /replacementAction>
  </template>
  <template name = “WriteAccessTracking”>
    <object> window </object>
    <property> alert </property>
    <states> s1, s2 </states>
  </template>
</policy>
```

Figure 8: Function replacement

A compound edit automaton can be obtained by combining read access filtering, call filtering, and write access tracking, as shown in Figure 8. Note that this is a direct combination of the three instantiated templates. This is in fact a very common pattern, because any methods could be redefined in incoming JavaScript code. Therefore, we use a compound template `FuncReplacement` (shown below) to directly represent this pattern. The actual policy on `window.alert` can then be obtained by instantiating this compound template with the corresponding parameters.

```
<template name = ”FuncReplacement”>
  <object> obj </object>
  <property> prop </property>
  <states> s1, s2 </states>
  <replacementAction> ... filtering code ... < /replacementAction>
</template>
```

We have also implemented some other compound templates to represent common ways of template composition. Examples include one on filtering default function calls up to redefinition and one on tracking the invocation of global event handlers.

As a summary, instantiated policy templates are expanded into edit automata, which in turn are compiled into rewriting rules and redirectors. The XML-based representation supports arbitrary edit automata for expressiveness, and some domain-specific templates are introduced to ease the task of policy construction.

body of the new definition would be instrumented as usual, if it involves relevant actions.

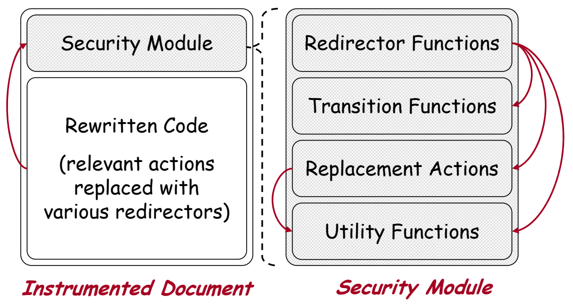


Figure 9: Structure of an instrumented document

5 Runtime Interaction with Proxy and with Client

5.1 Structure of instrumented documents

We have described two stand-alone components—one is the proxy for the instrumentation of incoming contents at runtime, the other is a policy compiler compiling high-level security policies (edit automata) into low-level rewriting rules off-line. These two components collaborate to complete the instrumentation tasks.

The structure of an instrumented document is shown in Figure 9. Based on the rewriting rules provided by the policy compiler, the proxy replaces relevant syntactic constructs in the incoming content with calls to redirectors. The proxy also inserts a *security module*, which contains the realization of the special instruction $instr$ and policy module Π in Figure 1. Specifically, $instr$ exhibits itself as a utility function `sec.instrument` (more in Section 5.2), and the policy module Π is interfaced through the redirectors. The redirectors call certain transition functions for maintaining edit automata at runtime, and invoke some replacement actions provided by policy designers. These functions may also refer to other utility functions as needed. We will discuss one such utility function on user notification in Section 5.3—upon a policy violation, `notifyPolicyViolation` is called to interact with the user effectively. Other commonly used utility functions include those for IP normalization, URL filtering, and policy object embedding and manipulation.

Since the security module is realized as regular JavaScript code, it resides in the same execution environment as incoming contents, and thus must be protected from malicious exploits. For example, malicious JavaScript code may modify the implementation of the security module by overwriting the functions of security checks. We organize the security module in a designated object named `sec`, and rename all other objects that could cause conflicts (*i.e.,* changing `sec` into `_sec`, `_sec` into `__sec`, and so on).

Although `sec` is usually inserted by the proxy during rewriting, sometimes a window could be created without going through the proxy at all. For example, the incoming content may use `window.open()` to open up an empty new window without triggering the proxy. Once the window is open, its content could be updated by the JavaScript code in the parent window. In response, we explicitly load a `sec` object into the empty new window when needed, so as to correctly enforce its future behaviors.

Another potential concern is that the incoming code could modify the call sites to the redirectors through JavaScript's flexible reflection mechanisms. Suppose the rewritten document contains a node that calls `sec.GetProp`. Using features such as innerHTML, incoming script could attempt to overwrite the node with some different code, such as `exploit`. This is in fact one form of higher-order script, and it is correctly handled, provably [21], following the theoretical framework of Figure 1. In particular, the runtime-generated script `exploit` will be sent back to the proxy for further instrumentation before executed; therefore, it cannot circumvent the security policy. In essence, the effect of using the runtime-generated code `exploit` as above is not different from statically using `exploit` in the original content directly.

5.2 Calling Proxy at Runtime

The utility function `instrument` needs to call the proxy at runtime to handle higher-order script. Therefore, we need to invoke the rewriters on the proxy from within the JavaScript code on the client. This is done with help of the `XMLHttpRequest` object [19] (or ActiveX in IE), which allows us to send runtime-generated JavaScript/HTML code to the proxy and receive their rewritten result. Although `XMLHttpRequest` is popularly known for its use in Ajax [7] (Asynchronous JavaScript and XML), we use it in a synchronous manner.

An interesting subtlety is that `XMLHttpRequest` is restricted to communicate only with servers that reside in the same domain as the origin of the current document. We use a specially encoded HTTP request, targeting the host of the current document, as the argument to `XMLHttpRequest`. Since all HTTP requests go through the proxy, the proxy is able to intercept relevant requests based on the path encoding and respond with the instrumentation result. Some key code, simplified (*e.g.,* the handling of the scope chain is omitted) from the actual implementation for ease of reading, is given below for illustration:

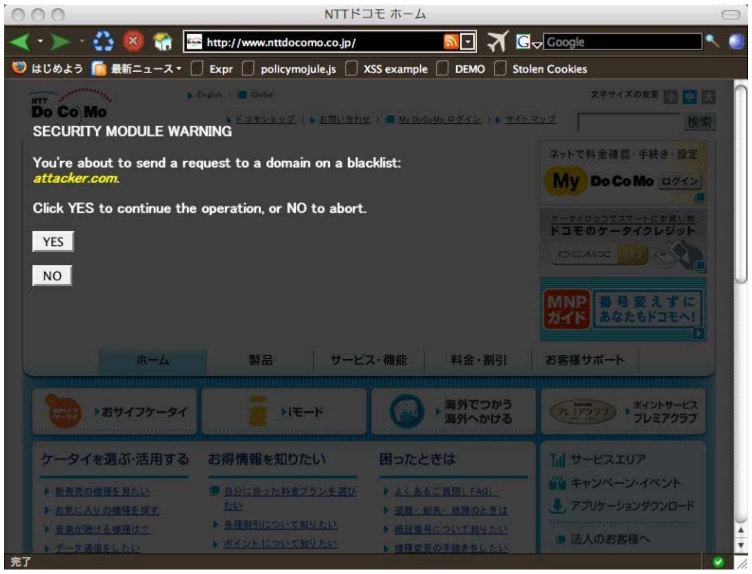
```
// This function sends str to the proxy for instrumentation.
// str is either HTML or JavaScript code, depending on type.
function instrument (type, str) {
  var xhr = new XMLHttpRequest();
  var url = "http://" + location.hostname + "/?__proxy__/"
                      + type + "&url = " + escape(location.href);
  try{
    xhr.open("POST", url, false);   // false specifies synchronous communication.
    xhr.send(str);
  }catch(e){...}
  return xhr.responseXML;           // The result of the instrumentation is in XML.
}
```

5.3 User Interaction

Effective user interaction upon a policy violation is important for practical deployment. A simple notification mechanism such as a dialogue box may not appear sufficiently friendly or informative to some users. Upon most policy violations, we overlay the notification messages on top of the rendered content (Figure 10). This better attracts the user's attention, disables the user's access to the problematic content, and allows the user to better assess the situation by comparing the notification message with the rendered content.

This would be straightforward if we were to change the browser implementation. However, in the proxy architecture, we need to implement such notification in HTML. To enable the overlaying effect, we use a combination of JavaScript and Cascading Style Sheets (CSS) to provide the desired font, color, visibility, opacity and rendering areas. An interesting issue occurs, however, because such functionality works by directly manipulating the HTML document tree. This manipulation normally happens after the entire document is loaded, *e.g.,* by using an `onload` event handler. Unfortunately, a policy violation may occur before the `onload` event happens. In this case, we apply one of the following "fall-back" mechanisms, based on different code and error scenarios.

The first is to notify the user immediately upon a policy violation using a regular dialogue box. Although less friendly looking than the overlaying approach, the blocking nature of the dialogue box desirably delays the rendering of any further content until receiving the user's instruction. A user could choose either to allow the action, to suppress the action and continue with the remainder content, or to stop rendering altogether. A slight variation of this is to directly stop the rendering of any further content and present error notification. This is mainly applicable to severe violations, such as upon identified phishing sites.



SECURITY MODULE WARNING

You're about to send a request to a domain on a blacklist:
attacker.com.

Click YES to continue the operation, or NO to abort.

YES

NO

Figure 10: Overlaid user notification upon a policy violation

The second is mainly applicable to relatively "mild" violations, such as pop-up windows. It suppresses the policy violating action, proceeds with the rendering of the remainder content, and presents the notification when the entire document is loaded. This allows us to present the notification in the friendly overlaid manner. However, there may be multiple actions suppressed when loading the content, thus the rendered content may not be as expected if not applied carefully.

The last is designed for special cases where the above are not suitable. For example, when a policy violation is caused by script within an inlined frame, the corresponding rendering area could be too small or even hidden, preventing the notification to be shown as expected. We thus present error messages by redirecting to a special web page supplied by the proxy.

5.4 Supporting Multiple Browsers

In theory, the proxy-centric architecture of Figure 3 is browser-independent, and the proxy should work naturally for all browsers. In practice, however, different browsers sometimes behave differently when interpreting the same JavaScript code. This is usually due to either implementation flaws or ambiguity and incompleteness of the JavaScript language specification [3] (*e.g.,* some behaviors are undefined).

First, different browsers have different tolerance on various types of obfuscated or malformed JavaScript and HTML code. For example, upon an assignment to the `window` object, IE throws an exception, Firefox, Konqueror and Safari ignore the assignment, and Opera allows the assignment. As another example, considering the following code: `x = document.write; x("...")`. Although triggering an exception on most browsers, this code will be rendered on IE. Our proxy does not currently handle such peculiarities correctly.

Next, some innocent-looking code may also result in different browser behaviors. Consider the following: `var location = "...";` . Whereas most browsers create a local variable shadowing the native `location` variable (the one holding the URL of the current document), Firefox ignores the local declaration of `var` and updates the native `location` variable instead.

Furthermore, different browsers support different DOM APIs. For instance, Firefox supports some convenient APIs such as `document.defaultView`, `document.__parent__`, `location.__parent__`, and `func.apply`. At some point, we used these APIs inside our security policies. Unfortunately, these APIs are not supported by some other browsers such as IE.

Table 3: Micro-benchmarks: Firefox rendering time (average)

	`Get`	`Call`	`GetD`	`CallD`	`Set`	`SetD`	`NewObj`	`ForIn`
time before (μs)	0.53	1.71	12.40	1.98	1.26	0.31	5.27	17.73
time after (μs)	9.44	36.81	100.57	23.65	240.75	33.37	37.75	66.40
ratio	17.8	21.5	8.1	11.9	191.1	107.6	7.2	3.7

Most of these browser-dependent behaviors are not of great importance to instrumentation. Nonetheless, it is useful to have the proxy behave consistently across different browsers. Therefore, we must instrument carefully using "robust" code—code that is treated uniformly by different browsers. Consider the (`FSrc`) template in Table 1 as an example. A space character is used as the rewritten value of the `src` attribute. If an empty string is used instead, some browsers will trigger the relevant event handler, but others will not.

In summary, it would be trivial having the proxy work with multiple browsers in an ideal world where all browsers behave in a standard way. In reality, however, one must address some browser-dependent issues.

6 Experiments

We have run our proxy with several browsers: Firefox, Konqueror, Opera, Safari, Safari Mobile on iPhone, and (partially) IE. During experiments, we manually confirmed that appropriate error notifications were given upon policy violations. In this section, we report some performance measurements on well-behaved web pages to demonstrate the overhead caused by the proxy. These measurements were made mainly using Firefox as the rendering browser, with help of the Firebug [8] add-on. Specifically, we profiled JavaScript execution in target web pages without counting in certain network-relevant activities, such as those due to the loading of inlined frames and external JavaScript sources, and the communication through `XMLHTTPRequest`. We have also conducted experiments using other browsers. Due to the lack of mature profiling tools for those browsers, we manually inserted timer functions in the target web pages. The resulting measurements were more fluctuating, but overall showed a fair match with the results on Firefox, and did not suggest specific new findings.

Two machines were used in the experiments—one as the proxy, the other as the client. Both machines have the same configuration: Intel Pentium 4, clock rate 3.2GHz, 1.5 GB of RAM, running FreeBSD 6.2.

6.1 Micro-benchmarks

In several rewriting cases of Table 3, simple JavaScript constructs are rewritten using redirectors. We expect this to be one major reason of performance penalties. Therefore, we crafted web pages to test such overhead on the browser with respect to the applicable rewriting cases. Specifically, we measured the browser rendering time before and after the instrumentation. Each case was measured by executing a minimal (thus the overhead is "exemplified") piece of code 7000-9000 times, and the entire process was repeated 10 times, whose average result is presented in Table 3. The slowdown ratios for Firefox ranged from under 10 to over 100. Note that these numbers are somewhat worst case scenarios, because in actual web pages, the cost will be amortized among additional computation that does not require nontrivial rewriting. The `set` case (property write) showed the biggest slowdown. By inspection, however, we did not see any significant structural difference in the rewritten code in comparison with the `get` case (property read).

There are a few rewriting cases not measured in the above micro-benchmarks, because they involve nontrivial interactions (*e.g.,* event triggering or network requests). The associated performance penalties for those cases are better measured and understood based on the macro-benchmarks shown next.

Table 4: Policy set and coverage of rewriting cases

	`Get`	`Call`	`GetD`	`CallD`	`Set`	`SetD`	`Event`	`FSrc`
Cookie	X							
IFrame					X			X
IP-URL					X	X		X
Pop-up	X	X	X	X	X	X		
URL-Event					X	X	X	X

Table 5: Target applications and various performance measurements

	DoCoMo *(corporate)*	LinkedIn *(social)*	WaMu *(bank)*	MSN *(portal)*	YouTube *(video)*	MSNBC *(news)*	GMap *(map)*	GMail *(email)*
size before (B)	24,433	97,728	156,834	170,927	187,324	404,311	659,512	899,840
size after (B)	28,047	144,646	141,024	252,004	232,606	495,568	959,097	1,483,577
ratio	1.15	1.48	0.90	1.47	1.24	1.23	1.45	1.65
proxy time (ms)	614	1,724	2,453	3,933	4,423	7,290	10,570	14,570
time before (ms)	94	82	553	402	104	2,832	1,369	4,542
time after (ms)	143	143	688	695	167	3,719	1,783	7,390
time ratio	1.52	1.74	1.24	1.73	1.61	1.31	1.30	1.63
time diff (ms)	49	61	135	293	63	887	414	2,848

6.2 Macro-benchmarks

To learn how the instrumentation works under typical browsing behaviors, we run macro-benchmarks using a selected set of policies and some popular web applications.

6.2.1 Policy Set

We crafted a set of policies to serve together as the policy input to the proxy, as listed in Table 4. The policies are selected based on both relevance to security and coverage of rewriting rules. The Cookie policy warns against the loading of dynamic foreign links after a cookie access [10], helping preventing XSS. The IFrame policy warns against foreign links serving as iframe sources, helping preventing a form of phishing. The IP-URL policy disallows dynamic IP URLs so as to prevent incoming script from analysing the presence of hosts on the local network. The Pop-up policy sets a limit on the number of pop-up windows, and restricts the behaviors of unwieldy (*e.g.,* , very small/large, out-of-boundary, and respawning) pop-ups. Finally, the URL-Event policy inspects certain event handlers (*e.g.,* `onclick`) to prevent malicious code from updating target URLs unexpectedly (*e.g.,* redirection to a phishing site after a linked is clicked). These together cover all the rewriting cases of Table 1. Note that the built-in rewriting rules of Table 2 are always applied regardless of what policy is in use.

6.2.2 Target Applications and Overheads

We hand-picked a variety of web pages as the target applications. These applications and their "code" sizes (for contents that require rewriting, including JavaScript and various HTML constructs, but excluding images, etc) before and after the instrumentation are listed in the top portion of Table 5. A variety of sizes are included, ranging from about 24KB to nearly 900KB. Recall that the proxy produces rewritten code and inserts a security module. The sizes in Table 5 are about the rewritten code only. The security module is always the same once we fix the policy set. For our policy set, the security module is 16,202 bytes; it is automatically cached by the browser, thus reducing network cost.

The ratio row shows how much the code size grew after instrumentation. In most cases, the growth was less than 50%. The worst case was the inbox of GMail, where the nearly 900KB of code grew by 65%. Interestingly, the WaMu code reduced by about 10% after instrumentation. By inspection, we found out that there was a significant amount of code commented out in the WaMu page, which was removed by the proxy.

The middle row of the table shows the time the proxy spent on rewriting. It was calculated based on the average of 50 runs. The figures are roughly proportional to the code sizes. The proxy spent a few seconds for smaller applications but over ten seconds for bigger ones. Since there was usually multiple code chunks processed, the client side perceptual delay was alleviated, because part of the content had started rendering even before the rewriting was done.

The bottom portion is Firefox interpretation time of the code before and after rewriting. The ratio and diff columns show the proportional and absolute time increases for rendering the instrumented code. For most applications, the absolute increase is negligible with respect to user perception. For GMail, the increase is nearly three seconds.

6.3 Safari Mobile on iPhone

Finally, we briefly report our experience of web browsing using Safari Mobile on iPhone. Although there has been some fluctuation of traffic in the WLAN, the numbers we collected still seems useful in describing the perceptual overhead introduced by the proxy.

When directly loading the applications of Table 5 without using the proxy, almost all pages started showing within 7-12 seconds, and finished loading (as indicated by a blue progress bar) within 11-24 seconds. The only exception was the inbox of GMail, which took well over a minute to be rendered. When loading the applications through the proxy, almost all pages started showing within 9-14 seconds (with exceptions described below). The finish time of the loading varied from 15 seconds to over a minute.

A few special cases are worth noting. MSNBC used a special page for rendering on iPhones; it took on average about 11 seconds for the entire page to be loaded without the proxy, and 15 seconds with the proxy. For GMap, we further experimented with the address searching functionality. It took on average 17 seconds to render the resulting map without the proxy, and 34 seconds with the proxy. When trying to render the inbox of GMail through the proxy, we got an error message of “method not allowed,” because the proxy does not currently support the CONNECT protocol [14] for tunnelling the SSL’ed login information. In contrast, our tested desktop browsers were set up to use direct connections for SSL’ed contents.

7 Related Work

There has been work applying code instrumentation to the security of machine code and Java bytecode. SFI [20] prevents access to memory locations outside of predefined regions. SASI [4] generalizes SFI to enforce security policies specified as security automata. Program Shepherding [11] restricts execution privileges on the basis of code origin, monitors control flow transfers to prevent the execution of data or modified code, and ensures that libraries are entered only through exported entry points. Naccio [6] enforces policies that place arbitrary constraints on resource manipulations as well as policies that alter how a program manipulates resources. PoET [5] applies inlined reference monitors to enforces EM policies [18] on Java bytecode. As pointed out by several studies [4, 17, 21], the above work is not directly applicable to JavaScript instrumentation for user-level security policies. Some notable problems include the pervasive use of reflection and higher-order script in JavaScript, the lack of flexible and usable policy management, and the difficulty of interpositioning a prototype-based object model.

The prototype in this paper is built based on the theoretical work of CoreScript instrumentation [21].

CoreScript focuses on formalizing the JavaScript execution model, the technique of instrumenting higher-order script, and the end-to-end proof of policy enforcement, and is largely abstract on the implementation aspects. Specifically on action rewriting, CoreScript has a designated syntactic category called actions, therefore syntax-directed rewriting is straightforward. This paper details the exact rewriting rules for various JavaScript syntactic constructs, introduces a policy compiler for generating syntactic rewriting rules from edit automata policies and templates, and discusses the subtle difficulties of proxy-based realization.

A closely related work is BrowserShield [17], which applies runtime instrumentation to rewrite HTML and JavaScript. Designed mainly as a vulnerability-driven filtering mechanism to defend browser vulnerabilities prior to patch deployment, the policies of BrowserShield are mainly about vulnerability signatures, and are written directly as JavaScript functions. After initial rewriting at an enterprise firewall, rewriting logic is injected into the target web page and executed at browser rendering time. A browser plug-in is used to enable the parsing and rewriting on the client.

In contrast, we target general user-level policies, and much of our work has been on practical policy management. Specifically, domain-specific abstractions are used for policy construction, and a policy compiler is engaged to translate such constructed policies to syntactic rewriting rules. Different syntactic categories are rewritten based on different policies. For simple policies, only one or two kinds of syntactic constructs are rewritten, although a composite policy typically requires the rewriting of more. All the rewriting (both load-time and run-time) happens on a proxy. The rewritten page interacts with the proxy at runtime using `XMLHttpRequest`. No software or plug-in is required on the client. Therefore, our architecture is naturally applicable to work with multiple web browsers, even if software/plug-in installation is not allowed.

JavaScript instrumentation has also been applied to the monitoring of client-side behaviors of web applications. Specifically, the AjaxScope [9] proxy applies on-the-fly parsing and instrumentation of JavaScript code as it is sent to users' browsers, and provides facilities for reducing the client-side overhead and giving fine-grained visibility into the code-level behaviors of the web applications. Targeting the development and improvement of non-malicious code (*e.g.,* during debugging), AjaxScope does not instrument code that is generated at runtime (*i.e.,* higher-order script). Therefore, the rewriting is simpler and, as is, not suitable as a security protection mechanism against malicious code.

8 Conclusion and Future Work

We have presented a JavaScript instrumentation prototype for browser security. Specifically, we reported experiences on instrumenting various JavaScript constructs, composing user-level security policies, and using a proxy to enable runtime rewriting. Our proxy enables flexible deployment scenarios. It naturally works with multiple browsers, and does not require software installation on the client. It provides a centralized control point for policy management, and poses relatively small computation requirement on the client. Although not optimized, our prototype yields promising results on the feasibility of the approach.

We plan to further extend the experimental work in the future. For example, we plan to conduct more experiments based on real-world web browsing patterns (*e.g.,* top URLs from web searches) and improve the support on popular browsers (most notably IE). We also plan to study the instrumentation of plug-in contents. For example, VBScript and Flash are both based on the ECMAScript standard [3], thus our instrumentation techniques are also applicable.

Our proxy-based architecture does not directly work with encrypted contents (*e.g.,* SSL). Some of our tested web pages (*e.g.,* GMail) uses SSL, but only for transmitting certain data, such as the login information. If the entire page was transmitted through SSL (as is the case of many banking sites), then the proxy cannot perform rewriting. This can be addressed by either decrypting on the proxy (if a trusted path between the proxy and the browser can be established), having the browser sending decrypted content back to the proxy, or directly implementing the instrumentation inside the browser.

Although the proxy-based architecture enables flexible deployment scenarios, a browser-based implementation may also be desirable. Some of our difficulties supporting multiple browsers have been due to their inconsistent treatment on undefined JavaScript behaviors. If implemented inside the browser, the same parsing process would be applied to both the rendering and the instrumentation, thus avoiding extra parsing overhead and problems caused by specific semantic interpretations.

References

[1] Apple Inc. Safari mobile on iphone. `http://www.apple.com/iphone/internet/`.

[2] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cve.mitre.org/`, 2007.

[3] ECMA International. ECMAScript language specification. Stardard ECMA-262, 3rd Edition, Dec. 1999.

[4] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. 1999 New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, Sept. 1999.

[5] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proc. IEEE S&P*, 2000.

[6] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. 20th IEEE S&P*, pages 32–47, 1999.

[7] J. J. Garrett. Ajax: A new approach to web applications. Adaptive Path essay, `http://www.adaptivepath.com/publications/essays/archives/000385.php`, Feb. 2005.

[8] J. Hewitt. Firebug—web development evolved. `http://www.getfirebug.com/`.

[9] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proc. SOSP'07*, pages 17–30, 2007.

[10] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proc. 2006 ACM Symposium on Applied Computing*, pages 330–337, 2006.

[11] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. 11th USENIX Security Symposium*, pages 191–206, 2002.

[12] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(2):2–16, Feb. 2005.

[13] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross-site scripting vulnerabilities in web applications. In *Proc. 6th IEEE International Workshop on Web Site Evolution*, pages 71–80, 2004.

[14] A. Luotonen. Tunneling TCP based protocols through web proxy servers. IETF RFC 2616, 1998.

[15] G. Ollmann. The phishing guide. `http://www.technicalinfo.net/papers/Phishing.html`.

[16] OWASP Foundation. The ten most critical web application security vulnerabilities. `http://www.owasp.org/`, 2007.

[17] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI'06*, Seattle, WA, 2006.

[18] F. B. Schneider. Enforceable security policies. *Trans. on Information & System Security*, 3(1):30–50, Feb. 2000.

[19] A. van Kesteren and D. Jackson. The XMLHttpRequest object. W3C working draft, `http://www.w3.org/TR/XMLHttpRequest/`, 2006.

[20] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. SOSP'93*, pages 203–216, Asheville, NC, 1993.

[21] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. POPL'07*, pages 237–249, Nice, France, Jan. 2007.