JavaScript Instrumentation for Browser Security

Dachuan Yu Ajay Chander Nayeem Islam Igor Serikov
DoCoMo Communications Laboratories USA, Inc.
{yu,chander,nayeem,iserikov}@docomolabs-usa.com

Abstract

It is well recognized that JavaScript can be exploited to launch browser-based security attacks. We propose to battle such attacks using program instrumentation. Untrusted JavaScript code goes through a rewriting process which identifies relevant operations, modifies questionable behaviors, and prompts the user (a web page viewer) for decisions on how to proceed when appropriate. Our solution is parametric with respect to the security policy—the policy is implemented separately from the rewriting, and the same rewriting process is carried out regardless of which policy is in use. Besides providing a rigorous account of the correctness of our solution, we also discuss practical issues including policy management and prototype experiments. A useful by-product of our work is an operational semantics of a core subset of JavaScript, where code embedded in (HTML) documents may generate further document pieces (with new code embedded) at runtime, yielding a form of self-modifying code.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Operational Semantics

General Terms Languages, Security, Theory

Keywords JavaScript, program instrumentation, edit automata, web browser

1. Introduction

JavaScript [4] is a scripting language designed for enhancing web pages. JavaScript programs are deployed in HTML documents and are interpreted by all major web browsers. They provide useful client-side computation facilities and access to the client system, making web pages more engaging, interactive, and responsive.

Unfortunately, the power and ubiquity of JavaScript has also been exploited to launch various browser-based attacks. On the one hand, there have been criminally serious attacks that steal sensitive user information, using, for example, techniques such as cross-site scripting (XSS) [11] and phishing [3]. On the other hand, there have been relatively benign annoyances which degrade the web-surfing experience, such as popping up advertising windows and altering browser configurations.

POPL'07 January 17–19, 2007, Nice, France

Aiming to thwart various attacks launched through JavaScript, we propose to regulate the behavior of untrusted JavaScript code using program instrumentation. Incoming script goes through a rewriting process which identifies and modifies questionable operations. At runtime, the modified script will perform necessary security checks and potentially generate user prompts, which are guided by a customizable security policy. Script that violates the policy will either be rewritten to respect the policy or stopped at runtime before the violation occurs unless the user decides to allow it. Compared with existing browser-security tools, which typically target specific attacks only, our approach enables a unified framework that enforces various security policies with the same underlying mechanism.

Although program instrumentation has been studied before, its application to JavaScript and browser security raises some interesting issues.

First, JavaScript has an interesting execution model where code, embedded in HTML documents, produces further HTML documents when executed. The produced HTML documents may also have code embedded in them. This gives rise to a form of self-modifying code which we refer to as *higher-order script* (as in script that generates other script). Higher-order script provides much expressiveness and is commonly used in many major websites. Unfortunately, it can also be exploited to circumvent a naïve rewriting mechanism. Previous work [20, 1] on the formal semantics of JavaScript subsets did not consider higher-order script, which limits its applicability. This work provides a rigorous characterization of higher-order script using an operational semantics of a subset of JavaScript that we call CoreScript. It is therefore a useful step toward understanding some tricky behaviors of JavaScript and preventing sophisticated exploits that employ higher-order script.

Next, we present the instrumentation of CoreScript programs as a set of formal rewriting rules. The rewriting is performed fully syntactically, even in the presence of higher-order script whose content is unknown statically. The runtime behavior of the instrumented code is confined by a security policy, in the sense that it calls a policy interface for bookkeeping and security checking for relevant operations. We show that our instrumentation is sound by proving that the observable security events in the execution of instrumented code respect the security policy.

We encode security policies using edit automata [10]. In particular, we make use of a policy interface for achieving a separation between policy management and the rewriting mechanism, identify what it takes for an edit automaton to reflect a sensible policy, and give a simple definition of policy combination that provably satisfies some basic requirements so as to be meaningful for our instrumentation.

The presentation and formal reasoning in this paper is based on an idealistic language CoreScript, which reflects the core features of JavaScript. Nonetheless, our techniques are applicable to the actual JavaScript language. There are some interesting issues when extending CoreScript instrumentation to support the com-

plete JavaScript language. We discuss how we have addressed those issues, thus bridging the gap between the CoreScript theory and its embodiment as a realistic security tool. We also describe a prototype implementation and some deployment experiments, which are based on the actual HTML and JavaScript.

We point out that the goal of this work is to explore provably safe protection against malicious client-side JavaScript code, assuming a correct browser implementation. This work is potentially also useful as a protection mechanism for temporarily patching security holes. However, no strong guarantee can be made without assuming that the browser interprets JavaScript code correctly.

The remainder of this paper is organized as follows. Section 2 provides background on JavaScript and browser security, outlines the difficulties and our approach, and compares with related work. Section 3 models CoreScript. Sections 4 and 5 present our policy framework and instrumentation method for regulating CoreScript. Section 6 bridges the gap between CoreScript and JavaScript. We describe our implementation and experiments in Sections 7 and 8, and conclude in Section 9.

2. Background

2.1 JavaScript Basics and Attacks

JavaScript is a popular tool for building web pages. JavaScript programs are essentially a form of mobile code embedded in HTML documents and executed on client machines. With help of the Document Object Model (DOM) [9] and other browser features, JavaScript programs can obtain restricted access to the client system and improve the functionality and appearance of web pages.

As is the case with other forms of mobile code, JavaScript programs introduce potential security vulnerabilities and loopholes for malicious parties to exploit. As a simple example, JavaScript is often used to open a new window on the client. This feature provides a degree of control beyond that offered by plain HTML alone, allowing the new window to have customized size, position, and chromes (*e.g.,* menu, tool bar, status bar). Unfortunately, this feature has been heavily exploited to generate annoying pop-ups of undesirable contents, some of which are difficult to "control" manually from a web user's point of view (*e.g.,* window-control buttons out of screen boundary, instant respawning when a window is closed). More seriously, this feature has also been exploited for launching phishing attacks [3], where key information about the origin of the web page is hidden from users (*e.g.,* a hidden location bar), and false information assembled to trick users into believing malicious contents (*e.g.,* a fake location bar).

As another example, JavaScript is often used to store and retrieve useful information (*e.g.,* a password to a web service) on the client machine as a "cookie." Such information is sometimes sensitive, therefore the browser restricts access to cookies based on the origin of web pages. For instance, JavaScript code from `attacker.com` will not be able to read a cookie set by `mybank.com`. Unfortunately, many web applications exhibit cross-site scripting (XSS) [11] vulnerabilities, where a malicious piece of script can be injected into a web page produced by a vulnerable server application. The browser interprets the injected script as if it were truly intended by the server application as benign code. As a result, the browser's origin-based protection is circumvented, and the malicious script may obtain access to the cookie set by the vulnerable application.

In general, JavaScript has been exploited to launch a wide range of attacks. A simple web search will reveal more details on this. The situation is potentially worse than for other forms of mobile code, *e.g.,* application downloading, because the user may not realize that loading web pages entails the execution of untrusted code.

```
<html>
  <head>...</head>
  <body>
    <p>
      <script>
      var name=parseName(document.cookie);
      document.write("Greetings, " + name + "!");
      </script>
      It is <em>great</em> to see you!
    </p>
  </body>
</html>
```

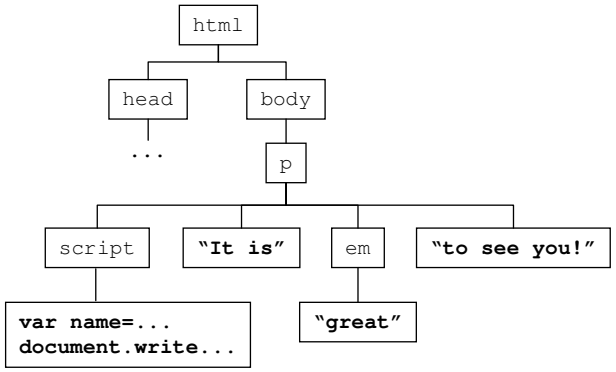

Figure 1. Script embedded in an HTML document

2.2 Difficulties and our Approach

We observe that many JavaScript vulnerabilities and threats can be addressed using program instrumentation. Properly inserted security checks and dialogue warnings can be used to identify and reveal to users potentially malicious behaviors. The extra computation overhead is usually acceptable, because JavaScript additions to web pages are typically small in size, and performance is not a major concern for most web pages in the context of user interactivity. Although the execution of JavaScript programs may follow complex control flows due to the interaction with users as well as with other documents (*e.g.,* in another window or frame), this is less of a concern for program instrumentation, because only very local analysis of the code is required.

Nonetheless, there remain some difficulties when applying existing program instrumentation techniques directly to JavaScript.

Execution model The execution model of JavaScript is quite different from that of other programming languages. A typical programming language takes input and produces output, possibly with some side effects produced during the execution. In the case of JavaScript, the program itself is embedded inside the "output" being computed. Figure 1 shows an example of a piece of script embedded in an HTML document. This script uses a function `parseName` (definition not shown) to obtain a user name from the cookie (`document.cookie`), and then calls a DOM API `document.write` to update the script node with some text.

Our CoreScript reflects this execution model. CoreScript programs are embedded in some tree-structured documents corresponding to HTML documents. In the operational semantics of CoreScript, script pieces are interpreted and replaced with the produced document pieces. The interpretation stops when no active script is present in the entire document.

Higher-order script The document pieces produced by JavaScript code at runtime could contain further JavaScript code. For example,

Script fragment:

```
var i = 1;
document.write("<script> i=2; document.write(i); </scr" + "ipt>" + i);
```

Output: 21

Script fragment:

```
var i = 1;
document.write("<script> i=2; document.write(i); </scr" + "ipt>");
document.write(i);
```

Output: 22

Figure 2. Execution order of higher-order script

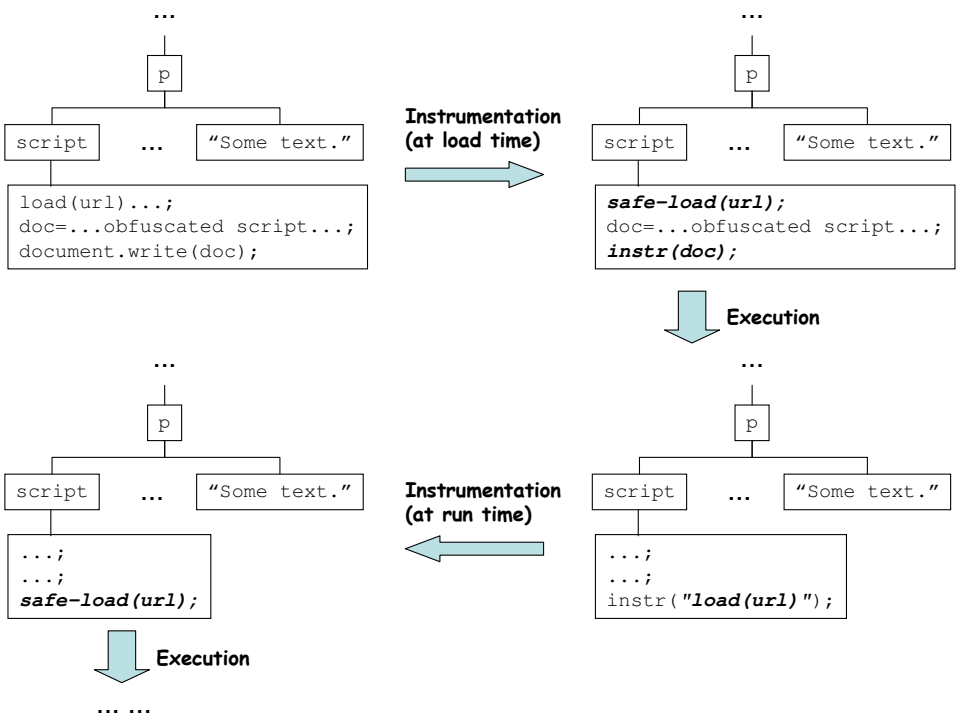

Figure 3. Instrumentation of higher-order script

the above DOM API `document.write` allows not only plain-text arguments, but also arbitrary HTML-document arguments that could possibly contain script nodes. These runtime-generated script nodes, when interpreted, may in turn produce more runtime-generated script nodes. In fact, infinite rounds of script generation can be programmed.

The behavior of an HTML document with higher-order script is sometimes hard to understand. For instance, the code fragments in Figure 2 appear to be similar, but produce different results. In this example, there are implicit conversions from integers to strings, + is string concatenation, and the closing tag `</script>` of the embedded script is intentionally separated so that the parser will not misunderstand it as a delimiter for the outer script fragment. The evaluation and execution order of higher-order script is not clearly specified in the language specification [4]. It is therefore useful to have a rigorous account as in CoreScript.

More importantly, higher-order script complicates instrumentation, because the instrumentation process cannot effectively identify all security events before the execution of the code—some events are embedded in string arguments which may not be apparent until runtime, such as computation results, user input, and documents loaded from a URL. In addition, there are many ways to obfuscate such embedded script against analyses and filters, *e.g.,* by a special encoding of some tag characters [8].

We handle the instrumentation of higher-order script through an extra level of indirection, as demonstrated in Figure 3. During the instrumentation, explicit security events such as `load(url)` will be directly rewritten with code that performs pertinent security checks and user warnings (abstracted as `safe-load(url)`). However, further events may be hidden in the generated script `doc`. Without inspecting the content of `doc`, which is a hardship statically, we feed `doc` verbatim to some special code `instr`. This

special code, when executed at runtime, will call back to the instrumentation process to perform the necessary inspection on the evaluation result of `doc`.

Such a treatment essentially delays some of the instrumentation tasks until runtime, making it happen on demand. We point out that the special code and other components of the instrumentation can be implemented either by changing the JavaScript interpreter in the browser or by using carefully written (but regular) JavaScript code which cannot be circumvented.

Policies and usability issues Some previous work [5] on program instrumentation for safety and security considers policies in the form of security automata [17], which stop program execution at runtime when a violation is detected. In the case of JavaScript programs, it is sometimes difficult to exactly characterize violations. For instance, suppose a web page tries to load a document from a different domain. This may be either an expected redirection or a symptom of an XSS attack. In such a situation, it is usually desirable to present pertinent information to the user for their discretion.

Our approach often modifies script to prompt the user about suspicious behaviors, as opposed to stopping the execution right away. When appropriate (*e.g.,* for out-of-boundary windows), we also carefully change the semantics of the code (*e.g.,* by "wrapping" the position arguments). We use edit automata [10], which are more expressive than security automata, to represent such policies. Naturally, our theorems for the correctness of the instrumentation are formulated with respect to edit automata.

Due to the wide use of JavaScript, it is difficult to provide a fixed set of policies for all situations. Thus, the ability to support customized policies is much desirable—there should be no change to the rewriting mechanisms when accommodating a new policy. We perform the same kind of rewriting regardless of the specifics of the policy; the rewriting produces code that refers to the policy through a fixed policy interface.

Furthermore, JavaScript and browser security is a mixture of many loosely coupled questions. A useful policy is hence typically a combination of multiple component policies. We take a simplified approach to policy combination, which is less expressive than some previous work [2], but effective for our desired protections.

The implementation of policy management can be carried out in a manner similar to the case of the special code for handling higher-order script, either by changing the JavaScript interpreter or by using regular JavaScript code. In the latter case, special care must be taken to ensure that the implementation cannot be circumvented.

Finally, we note that CoreScript is only a subset of JavaScript. This is necessary for better understanding the formal aspects and correctness of the approach. The generic treatment is also useful for potential application to other problem domains. Extending the solution to the entire JavaScript language or other implementations of ECMAScript [4] is mostly straightforward, but there are a few nontrivial aspects that deserve attention. We will discuss extensions and further issues in a later section.

2.3 Related Work

Browser security solutions JavaScript, DOM, and web browsers provide some basic security protections. Amongst the commonly used are sandboxing, same-origin policy, and signed scripting. Additional security extensions and coding styles are sometimes applied, such as the XPCNativeWrapper [12] that confines the access to certain methods and properties of untrusted objects. These only provide limited (coarse-grained) protections. There remain many opportunities for attacks, even if these protections are perfectly implemented. Representative example attacks that are not prevented by these include XSS, phishing, and resource abuse.

There have been also many separate browser security tools developed, such as pop-up blockers and SpoofGuard [3]. These separate solutions only provide protection against specific categories of attacks. In practice, it is sometimes difficult to deploy multiple solutions all together. In addition, there are many attacks that are outside of the range of protection of existing tools. Nonetheless, ideas and heuristics used in these tools are likely to be helpful for constructing useful security policies for instrumentation.

The above protection mechanisms are all deployed on the client side. Server-side protection has also been studied, especially in the context of command injection attacks [11, 18, 24]. The goal is to help well-intended programmers build web applications that are free of certain vulnerabilities, rather than to protect clients from malicious code.

Formal studies on JavaScript Existing work on formalizing JavaScript [20, 1] has focused on helping programmers write good code, as opposed to thwarting malicious exploits. On the technical front, the treatment of JavaScript programs used the conventional model, rather than as separate fragments embedded in HTML documents. Previous work also does not address the challenges posed by higher-order script.

Program instrumentation Program instrumentation has been much studied [22, 6, 5, 23, 15]. As detailed in the previous section, we extend program instrumentation in several ways to handle JavaScript, higher-order script, and some pragmatic policy issues. In addition, we provide end-to-end proofs on the correctness of our method: (1) instrumented programs will satisfy the security policy; (2) behavior of programs which satisfy the security policy will not be changed by the instrumentation. In constract, previous study [17, 10] on automata-based security policies did not address the integration of automata with programming languages.

3. CoreScript

We now model our subset of JavaScript—CoreScript. In particular, we give an operational semantics for CoreScript, focusing on higher-order script and its embedding in documents. Objects are omitted from this model, because they are orthogonal and have been carefully studied [20, 1]. Adding objects presents no additional difficulties for instrumentation.

3.1 Syntax

The syntax of CoreScript is given in Figure 4. In this figure, the symbols $[\,]$ are used as meta-parentheses, rather than as part of the CoreScript syntax.

A complete "world" W is a 4-tuple (Σ, χ, B, C). The first element Σ, a document bank, is a mapping from URLs l to documents D; this corresponds to the Internet. The second element χ, a variable bank, maps global variables x to documents D, and functions f to script programs P with formal parameters $\vec{x}$. The third element B, a browser, consists of possibly multiple windows; each window has a handle h for ease of reference, a document D as the content, and a domain name d marking the origin of the document. The fourth element C, a cookie bank, maps domain names to cookies in the form of documents (each domain has its own cookie). We use strings to model domain names d, paths p, and handles h. A URL l is a pair of a domain name d and a path p. We assume an implicit conversion between URLs and strings.

Documents D correspond to HTML documents. In JavaScript, all kinds of documents are embedded as strings using HTML tags such as `<script>` and `<em>`. They are treated uniformly as strings by all program constructs, but are parsed differently than plain strings when interpreted. Documents in CoreScript reflect this, except that we make different kinds of documents syntactically different, rendering the parsing implicit. A document is in either one of three syntactic forms: a plain string ($string$), a piece of

$$
\begin{array}{rrl}
(\textit{World}) & W & ::= (\Sigma, \chi, B, C) \\
(\textit{Document Bank}) & \Sigma & ::= \{[l = D]^*\} \\
(\textit{Variable Bank}) & \chi & ::= \{[x = D]^*, [f = (\vec{x})P]^*\} \\
(\textit{Browser}) & B & ::= \{[h = D \in d]^*\} \\
(\textit{Cookie Bank}) & C & ::= \{[d = D]^*\} \\
(URL) & l & ::= d.p \\
(Domain) & d & ::= \textit{string} \\
(Path) & p & ::= \textit{string} \\
(\textit{Handle}) & h & ::= \textit{string} \\
(Document) & D & ::= \textit{string} \mid \textit{js}\ P \mid F\ \vec{D} \\
(ValueDocument) & D^v & ::= \textit{string} \mid F\ \vec{D^v} \\
(Format) & F & ::= \texttt{jux} \mid \texttt{p} \mid \texttt{em} \mid \texttt{b} \mid \texttt{i} \mid \texttt{h1} \mid \texttt{h2} \mid \ldots \\
(Script) & P & ::= \texttt{skip} \mid x = E \mid P; P \\
 & & \mid \texttt{if}\ E\ \texttt{then}\ P\ \texttt{else}\ P \\
 & & \mid \texttt{while}\ E\ \texttt{do}\ P \mid f(\vec{E}) \mid \texttt{act}(A) \\
 & & \mid \texttt{write}(E) \\
(Expression) & E & ::= x \mid D \mid op(\vec{E}) \\
(Action) & A & ::= \epsilon \mid \texttt{newWin}(x, E) \mid \texttt{closeWin}(E) \\
 & & \mid \texttt{loadURL}(E) \mid \texttt{readCki}(x) \\
 & & \mid \texttt{writeCki}(E) \mid \texttt{secOp}(\vec{E}) \\
(Value\ Action) & A^v & ::= \epsilon \mid \texttt{newWin}(\_, D) \mid \texttt{closeWin}(D) \\
 & & \mid \texttt{loadURL}(D) \mid \texttt{readCki}(\_) \\
 & & \mid \texttt{writeCki}(D) \mid \texttt{secOp}(\vec{D})
\end{array}
$$

Figure 4. CoreScript syntax

script (*js* P), or a formatted document made up of a vector of sub-documents ($F\ \vec{D}$). Value documents D^v are documents that contain no script. We list a few common HTML format tags in the syntax as F, and introduce a new tag `jux` for the juxtaposition of multiple documents (this simplifies the presentation of the semantics).

The script programs P are mostly made up of common control constructs, including no-op, assignment, sequencing, conditional, while-loop, and function call. In addition, actions $\texttt{act}(A)$ are security-relevant operations that our instrumentation identifies and rewrites. Furthermore, higher-order script is supported using $\texttt{write}(E)$, where E evaluates at runtime to a document that may contain further script.

Expressions E include variables x, documents D, and other operations $op(\vec{E})$. Here we use op to abstract over all operations that are free of side-effects, such as string concatenation and comparison. We do not explicitly model booleans, instead simulating them with special documents (strings) `false` and `true`.

A few actions A are modeled explicitly for demonstration purposes. $\texttt{newWin}(x, E)$ creates a new window with E as the content document; a fresh handle is assigned to the new window and stored in x. $\texttt{closeWin}(E)$ closes the window which has handle E. $\texttt{loadURL}(E)$ directs the current window to load a new document from the URL E. $\texttt{readCki}(x)$ reads the cookie of the current domain into x. $\texttt{writeCki}(E)$ writes E into the cookie of the current domain. All other potential actions are abstracted as a generic $\texttt{secOp}(\vec{E})$. Value actions A^v are actions with document arguments only. Some arguments to actions are variables for storing results such as window handles or cookie contents. They are replaced with _ in value actions, because they do not affect the instrumentation.

3.2 Operational Semantics

We present the semantics of expressions in a big-step style in Figure 5. At runtime, expressions evaluate to documents, but not necessarily "value documents." As Rule (2) shows, D is not inspected for embedded script during expression evaluation. In Rule (3), we use $\hat{op}$ to refer to the corresponding meta-level computation of op.

$$\boxed{\chi \vdash E \Downarrow D}$$

$$\frac{}{\chi \vdash x \Downarrow \chi(x)} \quad (1)$$

$$\frac{}{\chi \vdash D \Downarrow D} \quad (2)$$

$$\frac{\chi \vdash \vec{E} \Downarrow \vec{D}}{\chi \vdash op(\vec{E}) \Downarrow \hat{op}(\vec{D})} \quad (3)$$

$$\boxed{\chi \vdash A \Downarrow A^v}$$

$$\frac{}{\chi \vdash \epsilon \Downarrow \epsilon} \quad (4)$$

$$\frac{\chi \vdash E \Downarrow D}{\chi \vdash \texttt{newWin}(x, E) \Downarrow \texttt{newWin}(\_, D)} \quad (5)$$

$$\frac{\chi \vdash E \Downarrow D}{\chi \vdash \texttt{closeWin}(E) \Downarrow \texttt{closeWin}(D)} \quad (6)$$

$$\frac{\chi \vdash E \Downarrow D}{\chi \vdash \texttt{loadURL}(E) \Downarrow \texttt{loadURL}(D)} \quad (7)$$

$$\frac{}{\chi \vdash \texttt{readCki}(x) \Downarrow \texttt{readCki}(\_)} \quad (8)$$

$$\frac{\chi \vdash E \Downarrow D}{\chi \vdash \texttt{writeCki}(E) \Downarrow \texttt{writeCki}(D)} \quad (9)$$

$$\frac{\chi \vdash \vec{E} \Downarrow \vec{D}}{\chi \vdash \texttt{secOp}(\vec{E}) \Downarrow \texttt{secOp}(\vec{D})} \quad (10)$$

Figure 5. Expression and action evaluation in CoreScript

Actions are evaluated to value actions as shown in the same figure. This process only computes the arguments of the actions, but does not actually perform the actions. Take Rule (9) as an example. The argument E is evaluated to D, but the cookie bank is not yet updated. It is also worth noting that, as indicated by the same rule, a cookie may be written using any document D, including a document with script embedded. Therefore, a program may store embedded script in a cookie for later use. Our instrumentation will be sound under this behavior.

We present the execution of a world in a small-step style in Figure 6. This is more intuitive when considering the security actions performed along with the execution, as well as their instrumentation.

Rules (11) and (12) define a multi-step relation. They refer to a single step relation as defined by Rule (13). This single step relation is non-deterministic, reflecting that any window could advance its content document at any time. Finally, Rule (14) uniformly advances the document in the window of handle h, with help of some macros defined in Figure 7.

The macro *focus* identifies the focus of the execution. It traverses a document and locates the left-most script component. The macro *stepDoc* computes an appropriate document for the next step, assuming that the focus of the argument document will be executed. The *focus* and *stepDoc* cases on value documents (*e.g.,* strings) are undefined. This indicates that nothing in value documents can be executed. If nothing can be executed in the entire document, then the execution terminates.

$focus(D) = P$
$stepDoc(D, \chi) = D'$

If $D =$	then $focus(D) =$	and $stepDoc(D, \chi) =$
$js\ P$ where $P \in \{\texttt{skip}, x = E, \texttt{act}(A)\}$	P	ε (empty string)
$js\ \texttt{write}(E)$	$\texttt{write}(E)$	D where $\chi \vdash E \Downarrow D$
$js\ P_1; P_2$	$focus(js\ P_1)$	$\texttt{jux}\ D\ (js\ P_2)$ where $D = stepDoc(js\ P_1, \chi)$
$js\ \texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2$	$\texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2$	$js\ P_1$ if $\chi \vdash E \Downarrow \texttt{true}$ $js\ P_2$ if $\chi \vdash E \Downarrow \texttt{false}$
$js\ \texttt{while}\ E\ \texttt{do}\ P$	$\texttt{while}\ E\ \texttt{do}\ P$	$js\ \texttt{if}\ E\ \texttt{then}\ (P; \texttt{while}\ E\ \texttt{do}\ P)\ \texttt{else skip}$
$js\ f(\vec{E})$	$f(\vec{E})$	$js\ P[\vec{D}/\vec{x}]$ where $\chi \vdash \vec{E} \Downarrow \vec{D}$ and $\chi(f) = (\vec{x})P$
$F\ \vec{D}^v D'\ \vec{D}$ where D' is not a value document	$focus(D')$	$F\ \vec{D}^v D''\ \vec{D}$ where $D'' = stepDoc(D', \chi)$

$step(P, h, W) = W'$

If $P =$	then $step(P, h, (\Sigma, \chi, B, C)) =$			
$\texttt{act}(\epsilon)$	$(\Sigma,\ \chi,$	$adv(B, h, \chi),$	$C)$	
$\texttt{act}(\texttt{newWin}(x, E))$	$(\Sigma,\ \chi\{x = h'\},$	$B'\{h' = D \in d\},$	$C)$	where $\chi \vdash E \Downarrow d.p$ $B' = adv(B, h, \chi)$ $\Sigma(d.p) = D$ h' is fresh
$\texttt{act}(\texttt{closeWin}(E))$	$(\Sigma,\ \chi,$	$B' - \{h'\},$	$C)$	where $\chi \vdash E \Downarrow h'$ $B' = adv(B, h, \chi)$
$\texttt{act}(\texttt{loadURL}(E))$	$(\Sigma,\ \chi,$	$B\{h = D \in d\},$	$C)$	where $\chi \vdash E \Downarrow d.p$ $\Sigma(d.p) = D$
$\texttt{act}(\texttt{readCki}(x))$	$(\Sigma,\ \chi\{x = C(d)\},$	$adv(B, h, \chi),$	$C)$	where $B(h) = D \in d$
$\texttt{act}(\texttt{writeCki}(E))$	$(\Sigma,\ \chi,$	$adv(B, h, \chi),$	$C\{d = D\})$	where $\chi \vdash E \Downarrow D$ $B(h) = D \in d$
$\texttt{act}(\texttt{secOp}(\vec{E}))$	...			
$x = E$	$(\Sigma,\ \chi\{x = D\},$	$adv(B, h, \chi),$	$C)$	where $\chi \vdash E \Downarrow D$
other P	$(\Sigma,\ \chi,$	$adv(B, h, \chi),$	$C)$	

where $adv(B, h, \chi) = B\{h = stepDoc(D, \chi) \in d\}$ if $B(h) = D \in d$

Figure 7. Helper functions of CoreScript semantics

$W \rightsquigarrow^* W' : \vec{A^v}$

$$\frac{}{W \rightsquigarrow^* W : \epsilon} \quad (11)$$

$$\frac{W \rightsquigarrow W' : A^v \qquad W' \rightsquigarrow^* W'' : \vec{A^v}}{W \rightsquigarrow^* W'' : A^v \vec{A^v}} \quad (12)$$

$W \rightsquigarrow W' : A^v$

$$\frac{W = (\Sigma, \chi, B, C) \qquad B = \{h_i = D_i \in d_i\}^{i=\{1\ldots n\}} \qquad \text{Pick any } j: \quad h_j \vdash W \rightsquigarrow W' : A^v}{W \rightsquigarrow W' : A^v} \quad (13)$$

$h \vdash W \rightsquigarrow W' : A^v$

$$\frac{B(h) = D \in d \qquad focus(D) = P \qquad step(P, h, (\Sigma, \chi, B, C)) = W \qquad A^v = \begin{cases} A_1^v & \text{if } P = \texttt{act}(A) \text{ and } \chi \vdash A \Downarrow A_1^v \\ \epsilon & \text{if } P \neq \texttt{act}(A) \end{cases}}{h \vdash (\Sigma, \chi, B, C) \rightsquigarrow W : A^v} \quad (14)$$

Figure 6. World execution in CoreScript

The macro *step* computes the step transition on worlds. Suppose we wish to make a step transition on world W by advancing the document in window h, and suppose the focus computation of the document in window h is P. The result world after the step transition would be $step(P, h, W)$. When defining *step*, the helper $adv(B, h, \chi)$ makes use of *stepDoc* to advance the document in window h.

It is worth noting that the semantics dictates the evaluation order for higher-order script, thus the two examples in Figure 2 are naturally explained. Take $\texttt{write}(op(\vec{E}))$; P as an example. CoreScript evaluates all E_i before executing the script embedded in any of them, explaining the behavior of the first script fragment in Figure 2. P is executed after the script generated by $\texttt{write}(op(\vec{E}))$ has finished execution, explaining the second.

4. Security Policies

The simple set of actions in CoreScript can already be exploited to launch browser-based attacks. For instance, one can open an unlimited number of windows (think pop-ups) and send sensitive cookie information to untrusted parties (think XSS). Various security policies can be designed to counter these attacks.

In our solution, policy management and code rewriting are two separate modules. Policies can be designed without knowledge of the rewriting process. A policy designer must ensure that the policies adequately reflect the desired protections. On the enforcement side, the rewriting process accesses the policy through a policy interface. The same rewriting mechanism is used for all policies.

This section describes the policy framework and presents the policy interface that the code rewriting of the next section uses.

$\boxed{\Pi \vdash \vec{A}}$

$$\frac{q \in \{\texttt{accept state}\}}{(Q, q, \delta) \vdash \epsilon} \tag{15}$$

$$\frac{\delta(q, A) = (q', A) \qquad (Q, q', \delta) \vdash \vec{A}}{(Q, q, \delta) \vdash A\vec{A}} \tag{16}$$

$\boxed{\Pi \vdash \vec{A} \Rightarrow \vec{A'}}$

$$\frac{}{\Pi \vdash \epsilon \Rightarrow \epsilon} \tag{17}$$

$$\frac{\delta(q, A) = (q', A') \qquad (Q, q', \delta) \vdash \vec{A} \Rightarrow \vec{A'}}{(Q, q, \delta) \vdash A\vec{A} \Rightarrow A'\vec{A'}} \tag{18}$$

Figure 8. Policy satisfaction and action editing

4.1 Policy Representation

Policies Π are expressed as edit automata [10]. An edit automaton is a triple (Q, q_0, δ), where Q is a finite or countably infinite set of states, $q_0 \in Q$ is the initial state (or current state), and δ is the complete transition function that has the form $\delta : Q * A \rightarrow Q * A$ (the symbol A is reused here to denote the set of actions in CoreScript). Note that δ may specify insertion, replacement, and suppression of actions, where suppression is handled by discarding the input action and producing an output action of ϵ. We require $\delta(q, \epsilon) = (q, \epsilon)$ so that policies are deterministic.

Figure 8 defines the meaning of a policy in terms of policy satisfaction (whether an action sequence is allowed) and action editing (how to rewrite an action sequence). Rules (15) and (16) define the satisfaction of a policy Π on an action sequence $\vec{A}$. Intuitively, $\Pi \vdash \vec{A}$ if and only if when feeding $\vec{A}$ into the automaton of Π, the automaton performs no modification to the actions, and stops at the end of the action sequence in a state that signals acceptance. In the remainder of this paper, we assume that every state is an "accept" state for simplicity, although it is trivial to relax this assumption.

Rules (17) and (18) define how a policy Π transforms an action sequence $\vec{A}$ into another action sequence $\vec{A'}$. Intuitively, $\Pi \vdash \vec{A} \Rightarrow \vec{A'}$ if and only if when feeding $\vec{A}$ into the automaton of Π, the automaton produces $\vec{A'}$.

Not all edit automata represent sensible policies. For instance, an edit automaton may convert action A_1 into A_2 and A_2 into A_1. It is unclear how an instrumentation mechanism should act under this policy, because even the recommended replacement action does not satisfy the policy. Therefore, it is useful to define the consistency of policies.

Definition 1 (Policy Consistency) A policy $\Pi = (Q, q_0, \delta)$ is consistent if and only if $\delta(q, A) = (q', A')$ implies $\delta(q, A') = (q', A')$ for any q, q', A and A'.

Theorem 1 (Sound Advice) Suppose Π is consistent. If $\Pi \vdash \vec{A} \Rightarrow \vec{A'}$, then $\Pi \vdash \vec{A'}$.

An inconsistent policy reflects an error in policy design. Syntactically, it is easy to convert an inconsistent policy into a consistent one: when the policy suggests a replacement action A' for an input action A under state q, we update the policy to also accept action A' under state q. More accurately, if $\delta(q, A) = (q', A')$, then we make sure $\delta(q, A') = (q', A')$. However, semantically, it is up to the policy designer to decide whether the updated policy is the intended one, especially in the case of conflicting updates. For instance, in the above example, the inconsistent policy may have already defined $\delta(q, A') = (q'', A'')$.

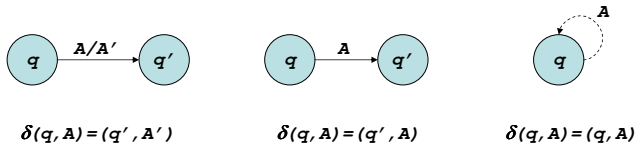

Figure 9. Edit automata as diagrams

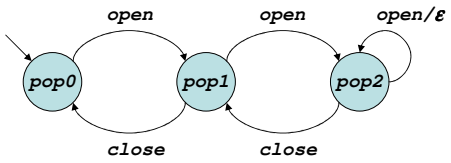

Figure 10. Automaton for a pop-up policy

We use consistent policies to guide our instrumentation, and policy consistency will serve as an assumption of our correctness theorems. Internally, a policy module maintains all states relevant to the edit automaton of the policy, including a current state and a complete transition function. Externally, the same policy module exposes the following interface to the rewriting process:

- Action review: $\texttt{check}(A)$.

This action review interface takes an input action as argument, advances the internal state of the automaton, and carries out a replacement action according to the transition function.

4.2 Policy Examples

The above policy framework is effective in identifying realistic JavaScript attacks and providing useful feedback to the user. We now demonstrate this with examples.

For ease of reading, we present edit automata as diagrams (Figure 9). To build a diagram from an edit automaton, we first create a node for every element of the state set. The node representing the starting state is marked with a special edge into the node. If the state transition function maps (q, A) into (q', A'), we add an edge from the node of q to the node of q', and mark the edge with A/A'. For conciseness, we use A to serve as a shorthand of A/A. If the state transition is trivial (performing no change to an input pair of state and action), we may omit that edge. Conversely, if a diagram does not explicitly specify an edge from state q with action A, there is an implicit A/A edge from the node of q to itself.

Figure 10 presents a policy for restricting the number of popup windows. The start state is $pop0$. State transition on $(pop0, close)$ is trivial (implicit). State transitions from the states with actions other than $open$ and $close$ are also trivial (implicit). This policy essentially ignores new window opening actions when there are already two pop-ups.

Figure 11 presents a policy for restricting the (potential) transmission of cookie information. The start state is *send-to-any*. In state *send-to-origin*, network requests are handled with a safe version of the loading action called *safe-loadURL*. In this policy, state transitions on *(send-to-any, loadURL(l)), (send-to-any, safe-loadURL), (send-to-origin, readCookie), (send-to-origin, safe-loadURL)* are trivial (implicit). State transitions from the states with actions other than reading, loading, and safe loading are also trivial (implicit). Essentially, this policy puts no restriction on loading before the cookie is read, but permits only safe loading afterwards.

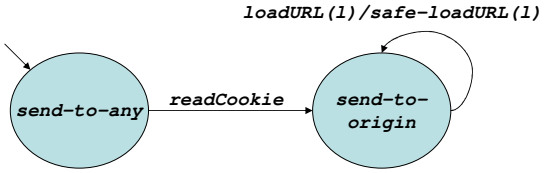

Figure 11. Automaton for a cookie policy

The implementation of the safe loading *safe-loadURL* performs necessary checks on the domain of the URL and asks the user whether to proceed with the loading if the domain of the URL does not match the origin of the document. We point out that, if desirable, a replacement action such as *safe-loadURL* may obtain information from the current state of the automaton and perform specialized security checks and user prompts. Its implementation is part of the policy module, and therefore does not affect the rewriting process. For now, it suffices to understand the implementation of safe actions as trusted and uncircumventable—safe actions are implemented correctly, and malicious script cannot overwrite their implementation.

4.3 Policy Combination

In practice, there are many different kinds of attacks. Naturally, there can be many different policies, each protecting against one kind of attack. Therefore, it is useful to combine multiple (without loss of generality, two) policies into one, which in turn guides the rewriting process.

For a policy combination ($\Pi_1 \oplus \Pi_2 = \Pi$) to be meaningful, it is sensible to require the following two conditions.

1. **Safe combination**: Suppose Π_1 and Π_2 are consistent. For all $\vec{A}, \Pi_1 \oplus \Pi_2 \vdash \vec{A}$ if and only if $\Pi_1 \vdash \vec{A}$ and $\Pi_2 \vdash \vec{A}$.
2. **Consistent combination**: If Π_1 and Π_2 are consistent, then $\Pi_1 \oplus \Pi_2$ is consistent.

We give a definition of policy combination which respects these requirements. Given two edit automata $\Pi_1 = (\{p_i | i = 0 \ldots n\}, p_0, \delta_1)$ and $\Pi_2 = (\{q_j | j = 0 \ldots m\}, q_0, \delta_2)$, we define:

$$\Pi_1 \oplus \Pi_2 = (\{p_i q_j | i = 0 \ldots n, j = 0 \ldots m\}, p_0 q_0, \delta)$$
$$\text{where } \delta(p_i q_j, A) = \begin{cases} (p_l q_k, A') & \text{if } \delta_1(p_i, A) = (p_l, A') \\ & \quad \text{and } \delta_2(q_j, A') = (q_k, A') \\ (p_l q_k, A') & \text{else if } \delta_2(q_j, A) = (q_k, A') \\ & \quad \text{and } \delta_1(p_i, A') = (p_l, A') \\ (p_i q_j, \epsilon) & \text{otherwise} \end{cases}$$

Intuitively, the combined policy simulates both component policies at the same time. When the first policy suggests an action that is agreed by the second policy, the combined policy takes that action. If not, it tries to see if the suggestion of the second policy is agreed by the first policy. In the worst case that neither of the above two holds, the combined policy suppresses the action. There is a combinatorial growth in the number of states after the combination. This does not pose a problem for an implementation, because a policy module may maintain separate state variables and transition functions for the component policies, yielding a linear growth in the policy representation. Based on the same reason, the above definition extends naturally to support countably finite-state component policies.

It is easy to check that this definition of combination satisfies the above safety and consistency requirements. Nonetheless, we point out that there exist other sensible definitions of combination that also satisfy the same requirements. For example, the above definition "prefers" the first policy over the seond. A similar definition that prefers the second is also sensible. Furthermore, a more sophisticated combination may attempt to resolve conflicts by recursively feeding suggested actions into the automata, whereas the above simply gives up after the first try. Note that the requirement of "safe combination" only talks about acceptable action sequences, not about replacement actions. In general, related study on policy combination [16, 2] may provide some alternatives.

Based on our experience, the above definition of combination seems to work well in practice. Many policies in our experiments, such as the two examples shown earlier, are orthogonal to each other in the sense that they deal with separate sets of actions. Their combination using the above definition is straightforward as expected. Our "unorthogonal" policies do not suggest contradicting replacement actions. We suspect that if two policies are conflicting (*i.e.,* the "otherwise" case in the above definition), it is likely a design error, and "divine intervention" from the policy designer would be best.

5. CoreScript Instrumentation

Given the policy module and its interface in the previous section, the instrumentation of CoreScript becomes a straightforward syntax-directed rewriting process. We now present the rewriting and its correctness.

The task of the rewriting process is to traverse the document tree and redirect all actions through the policy module. Whenever an action $\texttt{act}(A)$ is identified, we redirect it to the action interface $\texttt{check}(A)$, trusting that the policy module will carry out an appropriate replacement action at runtime. For higher-order script $\texttt{write}(E)$, we feed the document argument E verbatim to a special interface $\texttt{instr}(E)$, whose implementation will call back to the rewriting process at runtime after E is evaluated.

In this section, we organize the above two interfaces as two new CoreScript instructions for the rewriting process to use. In particular, we extend the syntax of CoreScript as follows.

$$(Script) \quad P ::= \ldots \mid \texttt{instr}(E) \mid \texttt{check}(A)$$

The details of the rewriting are given in Figure 12. Its simplicity is obvious—no knowledge is required on the meaning or the implementation of the two new instructions. The nontrivial tasks are performed by Rules (19) and (20), where the new instructions are used to replace runtime code generation and actions. All other rules simply propagate the rewriting results. We give the rewriting cases for the two new instructions in Rule (24), which allows the rewriting to work on code that calls the two interfaces. We also define the rewriting on world W and its four components. In an implementation, some components (*e.g.,* the document bank Σ) will be instrumented on demand (*e.g.,* when loaded).

We give the semantics of the two new instructions so as to reason about the correctness of the instrumentation. For $\texttt{instr}(E)$, the purpose is to mark script generation and delay the instrumentation until runtime. Therefore, its operational semantics should evaluate the argument expression and feed it through the rewriting process. The following definitions capture exactly that.

$$\begin{array}{l} \mathit{focus}(\mathit{js}\ \texttt{instr}(E)) = \texttt{instr}(E) \\ \mathit{stepDoc}(\mathit{js}\ \texttt{instr}(E), \chi) = \iota(D) \quad \text{where } \chi \vdash E \Downarrow D \\ \mathit{step}(\texttt{instr}(E), h, (\Sigma, \chi, B, C)) = (\Sigma, \chi, \mathit{adv}(B, h, \chi), C) \end{array} \tag{33}$$

Recall that adv is defined in Figure 7. The operational semantics rules for other language constructs remain the same under the addition of `instr`. The *focus* and *step* function cases defined above fit in well with Rule (14), which makes a step on a document given a specific window handle.

$\boxed{\iota(P) = P'}$

$$\frac{}{\iota(\texttt{write}(E)) = \texttt{instr}(E)} \tag{19}$$

$$\frac{}{\iota(\texttt{act}(A)) = \texttt{check}(A)} \tag{20}$$

$$\frac{}{\iota(P_1; P_2) = \iota(P_1); \iota(P_2)} \tag{21}$$

$$\frac{}{\iota(\texttt{if } E \texttt{ then } P_1 \texttt{ else } P_2) = \texttt{if } E \texttt{ then } \iota(P_1) \texttt{ else } \iota(P_2)} \tag{22}$$

$$\frac{}{\iota(\texttt{while } E \texttt{ do } P) = \texttt{while } E \texttt{ do } \iota(P)} \tag{23}$$

$$\frac{P \in \{\texttt{skip}, x{=}E, f(\vec{E}), \texttt{instr}(E), \texttt{check}(A)\}}{\iota(P) = P} \tag{24}$$

$\boxed{\iota(D) = D'}$

$$\frac{}{\iota(string) = string} \tag{25}$$

$$\frac{}{\iota(js\ P) = js\ \iota(P)} \tag{26}$$

$$\frac{\iota(\vec{D}) = \vec{D'}}{\iota(F\ \vec{D}) = F\ \vec{D'}} \tag{27}$$

$\boxed{\iota(\Sigma) = \Sigma'}$

$$\frac{}{\iota(\{(l = D)^*\}) = \{(l = \iota(D))^*\}} \tag{28}$$

$\boxed{\iota(\chi) = \chi'}$

$$\frac{}{\iota(\{[x = D]^*, [f = (\vec{x})P]^*\}) = \{[x = \iota(D)]^*, [f = (\vec{x})\iota(P)]^*\}} \tag{29}$$

$\boxed{\iota(B) = B'}$

$$\frac{}{\iota(\{(h = D \in d)^*\}) = \{(h = \iota(D) \in d)^*\}} \tag{30}$$

$\boxed{\iota(C) = C'}$

$$\frac{}{\iota(\{[d = D]^*\}) = \{[d = \iota(D)]^*\}} \tag{31}$$

$\boxed{\iota(W) = W'}$

$$\frac{}{\iota((\Sigma, \chi, B, C)) = (\iota(\Sigma), \iota(\chi), \iota(B), \iota(C))} \tag{32}$$

Figure 12. Syntax-directed rewriting

An inspection of Rule (33) will show that the rewriting process ι is called at run time after evaluting E to D. The execution of ι always terminates, producing an instrumented document. In this instrumented document, there is potentially further hidden script marked by further `instr`. Such hidden script will be rewritten later when it is generated.

We define the semantics of $\texttt{check}(A)$ in a similar fashion using the following definitions.

$$\begin{array}{l} focus(js\ \texttt{check}(A)) = \texttt{check}(A) \\ stepDoc(js\ \texttt{check}(A), \chi) = \epsilon \\ step(\texttt{check}(A), h, (\Sigma, \chi, B, C)) = \textit{undefined} \end{array} \tag{34}$$

The *focus* case for $\texttt{check}(A)$ is trivially $\texttt{check}(A)$ itself. The execution of $\texttt{check}(A)$ will consume $\texttt{check}(A)$ entirely and leave no further document piece for the next step, hence the *stepDoc* case. The *step* case is undefined, because we will never refer to this case in the updated operational semantics.

With the addition of `check`, the program execution is connected to the policy module. Therefore, in the updated operational semantics, we need to take into account the internal state of the policy module (the state of the edit automaton). We extend the previous reduction relations of CoreScript in Figure 13, where the new formations of the reduction relations explicitly specify the automaton transition function (δ) and the automaton states (q and q'). Similar to the previous semantics, the multi-step relation defined by Rules (35) and (36) is a reflexive and trasitive closure of a non-deterministic step relation defined by Rule (37). This non-deterministic step relation is defined with help of a determinstic step relation, which we call "document advance."

$\boxed{\vdash_\delta (W, q) \rightsquigarrow^* (W', q') : \vec{A^v}}$

$$\frac{}{\vdash_\delta (W, q) \rightsquigarrow^* (W, q) : \epsilon} \tag{35}$$

$$\frac{\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v \qquad \vdash_\delta (W', q') \rightsquigarrow^* (W'', q'') : \vec{A^v}}{\vdash_\delta (W, q) \rightsquigarrow^* (W'', q'') : A^v \vec{A^v}} \tag{36}$$

$\boxed{\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v}$

$$\frac{\begin{array}{l} W = (\Sigma, \chi, B, C) \qquad B = \{h_i = D_i \in d_i\}^{i=\{1\ldots n\}} \\ \text{Pick any } j: \qquad h_j \vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v \end{array}}{\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v} \tag{37}$$

$\boxed{h \vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v}$

$$\frac{\begin{array}{l} B(h) = D \in d \qquad focus(D) \neq \texttt{check}(A) \\ h \vdash (\Sigma, \chi, B, C) \rightsquigarrow W : A^v \end{array}}{h \vdash_\delta ((\Sigma, \chi, B, C), q) \rightsquigarrow (W, q) : A^v} \tag{38}$$

$$\frac{\begin{array}{l} B(h) = D \in d \qquad focus(D) = \texttt{check}(A) \\ \chi \vdash A \Downarrow A_1^v \qquad \delta(q, A_1^v) = (q', A^v) \\ step(\texttt{act}(A^v), h, (\Sigma, \chi, B, C)) = W \end{array}}{h \vdash_\delta ((\Sigma, \chi, B, C), q) \rightsquigarrow (W, q') : A^v} \tag{39}$$

Figure 13. World execution in CoreScript with policy module

Document advance is defined by Rules (38) and (39). When the focus of the document is not a call to `check`, the old document advance relation (defined in Rule (14)) is used, and the automaton state remains unchanged. When the focus is a call to `check`, the automaton state is updated and the replacement action is produced according to the transition function, and the world components are updated using the step case of $\texttt{act}(A^v)$ because the replacement action A^v is carried out instead of the original action A.

We have now completed the updated semantics. Essentially, a policy instance is executed alongside the program execution—the current state of the policy instance is updated in correspondence with the actions of the program.

5.1 Correctness Theorems

We present the correctness of the instrumentation as two theorems—soundness and transparency [10]. Soundness states that instrumented code will respect the policy. Transparency states that the instrumentation will not affect the behavior of code that already respects the policy. The intuition behind these correctness theorems is straightforward, since our instrumentation feeds all actions through the policy module for suggestions. Soundness holds because the suggested actions always satisfy the policy due to policy consistency. Transparency holds because the suggested actions would be identical to the input actions if the input actions already satisfy the policy. In what follows, we establish these two theorems with a sequence of lemmas.

First, we introduce a notion of orthodoxy.

Definition 2 (Orthodoxy) W (or Σ, χ, B, C, D, P) is orthodox if it has no occurrence of $\texttt{act}(A)$ or $\texttt{write}(E)$.

It is easy to see that our instrumentation produces orthodox results, as in the following lemma.

Lemma 1 (Instrumentation Orthodoxy) $\iota(P)$, $\iota(D)$, $\iota(C)$, $\iota(B)$, $\iota(\chi)$, $\iota(\Sigma)$, and $\iota(W)$ are orthodox.

Proof sketch: By simultaneous induction on the structures of P and D. By case analysis on the structures of C, B, χ, Σ, and W. □

We show that orthodoxy is preserved by the step relation, as follows.

Lemma 2 (Orthodoxy Preservation) If W is orthodox and $\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v$, then W' is orthodox.

Proof sketch: By definition of the step relation ($\rightsquigarrow$), with induction on the structure of documents. The case of executing $\texttt{write}(E)$ is no possible because W is orthodox. In the case of executing $\texttt{instr}(E)$, the operational semantics produces an instrumented document to replace the focus node. Orthodoxy thus follows from Lemma 1. In all other cases, the operational semantics may obtain document pieces from other program components, which are orthodox by assumption. □

The execution of an orthodox world always respects the policy, as articulated below.

Lemma 3 (Policy Satisfaction) Suppose $\Pi = (Q, q, \delta)$ is consistent. If W is orthodox and $\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v$, then $\delta(q, A^v) = (q', A^v)$.

Proof sketch: By case analysis on the step relation ($\rightsquigarrow$). In the case of executing $\texttt{check}(A)$, by inversion on Rule (39), $\delta(q, A_1^v) = (q', A^v)$. The expected result $\delta(q, A^v) = (q', A^v)$ follows directly from the definition of policy consistency. In all other cases, by inversion on Rule (38), $q = q'$. By further inversion on Rule (14), we get $A^v = \epsilon$ (the case of executing $\texttt{act}(A)$ is not possible because W is orthodox). $\delta(q, \epsilon) = (q, \epsilon)$ because of the deterministic requirement on policies. □

The soundness theorem follows naturally from these lemmas.

Theorem 2 (Soundness) Suppose $\Pi = (Q, q, \delta)$ is consistent. If W is orthodox and $\vdash_\delta (W, q) \rightsquigarrow^* (W', q') : \vec{A^v}$, then $\Pi \vdash \vec{A^v}$.

Proof sketch: By structural induction on the multi-step relation ($\rightsquigarrow^*$). The base case of zero step and empty output action is trivial. In the inductive case, there exists W_1, q_1 and A_1^v such that $\vdash_\delta (W, q) \rightsquigarrow (W_1, q_1) : A_1^v$, $\vdash_\delta (W_1, q_1) \rightsquigarrow^* (W', q') : \vec{A^v}'$, and $\vec{A^v} = A_1^v \vec{A^v}'$. By Lemma 3, $\delta(q, A^v) = (q_1, A^v)$. (Q, q_1, δ) is consistent by assumption and definition of policy consistency. W_1 is orthodox by Lemma 2. By induction hypothesis, $(Q, q_1, \delta) \vdash \vec{A^v}'$. By definition of policy satisfaction, $\Pi \vdash \vec{A^v}$. □

From the instrumentation's perspective, it is desirable to establish that $\iota(W)$ is safe given any W. This follows as a corollary of Theorem 2, because $\iota(W)$ is orthodox by Lemma 1.

To formulate the transparency theorem, we use the multi-step relation defined in Section 3 before the instrumentation extension. This reflects the intuition that incoming script should be a sensible CoreScript (or JavaScript) program without knowledge about the policy module. We first introduce a lock step lemma to relate the single-step execution of instrumented code with the single-step execution of the original code in the case where the original code satisfies the policy.

Lemma 4 (Lock step) If $W \rightsquigarrow W' : A^v$ and $\delta(q, A^v) = (q', A^v)$, then $\vdash_\delta (\iota(W), q) \rightsquigarrow (\iota(W'), q') : A^v$.

Proof sketch: By definition of the step relation ($\rightsquigarrow$), with induction on the structure of documents. The focus of $\iota(W)$ refers to a tree node in correspondence with the focus of W.

In the case that $\texttt{write}(E)$ is the focus of W, $\texttt{instr}(E)$ will be the focus of $\iota(W)$. The operational semantics of `write` and `instr` carry out a similar evaluation on the argument E, except that $\texttt{instr}(E)$ uses an instrumented variable environment and returns an instrumented result document. The output action A^v is ϵ in both cases. We can construct the derivation of $\vdash_\delta (\iota(W), q) \rightsquigarrow (\iota(W'), q') : A^v$ by: (i) following Rule (37) and choosing the same handle h as used for obtaining $W \rightsquigarrow W' : A^v$; (ii) following Rule (38), which refers back to the old single-step relation $h \vdash \iota(W) \rightsquigarrow \iota(W') : A^v$; then (iii) following the derivation of $h \vdash W \rightsquigarrow W' : A^v$ used for obtaining $W \rightsquigarrow W' : A^v$, with various components replaced with the instrumented version.

In the case that $\texttt{act}(A)$ is the focus of W, $\texttt{check}(A)$ will be the focus of $\iota(W)$. `act` and `check` both produce an empty string to replace the focus tree node. The operational semantics of $\texttt{act}(A)$ will evaluate A to A^v (Rule (14)). The operational semantics of $\texttt{check}(A)$ will evaluate A to A^v and feed A^v to the policy (Rule (39)). By assumption, $\delta(q, A^v) = (q', A^v)$. Therefore, `act` and `check` will produce the same output action in this case. The operational semantics of $\texttt{check}(A)$ will further apply the macro *step* to $\texttt{act}(A^v)$ to update the world components. Therefore, further derivations of the two reductions follow the same structure.

In all other cases, W and $\iota(W)$ will be executing the same instructions. The derivation of the instrumented reduction follows that of the original reduction. □

The transparency theorem follows naturally from the lock step lemma.

Theorem 3 (Transparency) If $W \rightsquigarrow^* W' : \vec{A^v}$ and $(Q, q, \delta) \vdash \vec{A^v}$, then $\vdash_\delta (\iota(W), q) \rightsquigarrow^* (\iota(W'), q') : \vec{A^v}$.

Proof sketch: By structural induction on the multi-step relation ($\rightsquigarrow^*$). The base case of zero step and emtpy output action is trivial. In the inductive case, there exists W_1 and A_1^v such that $W \rightsquigarrow W_1 : A_1^v$, $W_1 \rightsquigarrow^* W' : \vec{A^v}'$, and $\vec{A^v} = A_1^v \vec{A^v}'$. By assumption $(Q, q, \delta) \vdash \vec{A^v}$ and definition of policy satisfaction, there exists q_1 such that $\delta(q, A_1^v) = (q_1, A_1^v)$ and $(Q, q_1, \delta) \vdash \vec{A^v}'$. By Lemma 4, $\vdash_\delta (\iota(W), q) \rightsquigarrow (\iota(W'), q_1) : A_1^v$. By induction hypothesis, $\vdash_\delta (\iota(W_1), q_1) \rightsquigarrow^* (\iota(W'), q') : \vec{A^v}'$. By Rule (36), $\vdash_\delta (\iota(W), q) \rightsquigarrow^* (\iota(W'), q') : \vec{A^v}$. □

In the above transparency theorem, the original world W does not refer to the instrumentation and policy interfaces, reflecting that incoming script is written in regular JavaScript. We can also formulate a variant of the transparency theorem to allow incoming script that refers to the instrumentation and policy interfaces, as follows.

Theorem 4 (Extended Transparency) If $\vdash_\delta (W, q) \rightsquigarrow^* (W', q') : \vec{A^v}$ and $(Q, q, \delta) \vdash \vec{A^v}$, then $\vdash_\delta (\iota(W), q) \rightsquigarrow^* (\iota(W'), q') : \vec{A^v}$.

This theorem allows W to be unorthodox—W may contain a mixture of `write`, `act`, `instr` and `check`. The proof of this theorem requires a similarly extended lock-step lemma. The proof extension is straightforward, because on the two new cases allowed by this theorem (`instr` and `check`), the rewriting is essentially an identity function.

6. Discussions

We have modeled CoreScript as a core language for client-side scripting. Its distinguishing features include the embedding of script in documents, the generation of new script at runtime, and distinguishing security-relevant actions. CoreScript abstracts away some specific details of JavaScript so that the ideas are applicable to other browser-based scripting languages. Nonetheless, any practical realization of the approach will have to tackle some more language-specific details.

First, CoreScript supports the embedding of code in a document tree using *js* nodes. Such a treatment is adapted from the use of `<script>` tags in JavaScript (Figure 1 provided an example). Beyond the `<script>` tags, there are many other ways for embedding script in an HTML document. Some common places where script could occur include images (*e.g.,* `<IMG SRC=...>`), frames (*e.g.,* `<IFRAME SRC=...>`), XML (*e.g.,* `<XML SRC=...>`, tables (*e.g.,* `<TABLE BACKGROUND=...>`), and body background (*e.g.,* `<BODY BACKGROUND=...>`. Furthermore, script can also be embedded in a large number of event handlers (*e.g.,* `onActivate()`, `onClick()`, `onLoad()`, `onUnload()`, ...). A realization of our approach must also identify and rewrite such embedded script.

Second, CoreScript makes use of `write`(E) to generate script at runtime. This is a unified view on several related functions, including `eval` in the JavaScript core language and `window.execScript`, `document.write`, `document.writeln` in the DOM. These functions all take string arguments. `eval` evaluates a string as a JavaScript statement or expression and returns the result. In contrast, `window.execScript` executes one or more script statements but returns no values. CoreScript's treatment on higher-order script is expressive enough for these two.

However, `document.write` and `document.writeln` are more challenging. These two functions send strings as document fragments to be displayed in their windows, where the document fragments could have script embedded. Interestingly, these document fragments do not have to be complete document tree nodes. Instead, they can be pieced together with other strings to form a complete node, as demonstrated in the following examples.

```
<script>
document.write("<scr");
document.write("ipt> malic");
var i = 1;
document.write("ious code; </sc");
document.write("ript>");
</script>

<script>
document.write("<scr");</script>ipt>
malicious code
</script>
```

Each of the above `write` would appear to produce harmless text to a naïve filter. To avoid such loopholes when applying CoreScript instrumentation, one possibility is to piece together generated document fragments before feeding them into the rewriting process of the next stage. This must be done with care to avoid changing the semantics of the code (recall Figure 2). Observing that the expressiveness of producing new script as broken-up fragments does not seem to be useful in well-intended programs, a better solution might be to simply disrupt the generation of ungrammatical script pieces. As an example, Su and Wassermann [18] use metacharacters to delimit user input syntactically and prevent command injection attacks. A similar technique can be applied here to delimit generated document pieces implicitly and prevent the above kind of attacks (the generated document pieces "`<scr`" and "`ipt>`" will no longer be pieced together to form a script tag, if they are generated in two separate "`document.write`" statements).

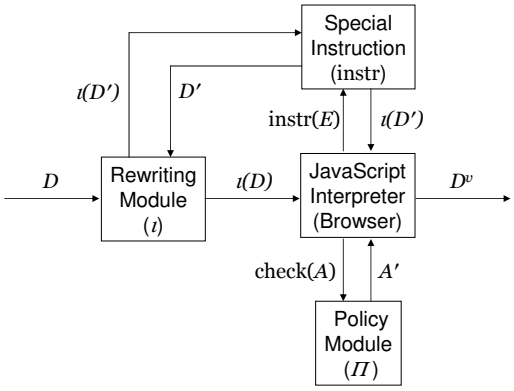

Figure 14. Implementation architecture

CoreScript does not provide a means to modify the content of a document in arbitrary ways, because a `write`(E) node generates a new node to be positioned at the exact same place in the document tree. The DOM provides other means for modifying a document. For instance, a document could be modified through the `innerHTML`, `innerText`, `outerHTML`, `outerText`, and `nodeValue` properties of any element. These are not covered in the CoreScript model. Nonetheless, an extension is conceivable, where the mechanisms for runtime script generation specifies which node in the document tree is to be updated. The instrumentation method remains the same, because it does not matter where the generated script is located, as long as it is rewritten appropriately to go through the policy interface.

Lastly, CoreScript includes some simple actions for demonstration purposes. A realization would accommodate many other actions pertinent to attacks and protections. Some relevant DOM APIs include those for manipulating cookies, windows, network usage, clipboard, and user interface elements. In addition, it is useful to introduce implicit actions for some event handlers. For instance, the "undead window" attack below could be prevented by disallowing window opening inside an `onUnload` event.

```
<html>
  <head>
    <script type="text/javascript">
      function respawn(){
          window.open("URL/undead.html")
      }
    </script>
  </head>
  <body onunload="respawn()">
    Content of undead.html
  </body>
</html>
```

7. Implementation

As shown in Figure 14, our implementation extends a browser with three small modules—one for the syntactic code rewriting (ι), one for interpreting the special instruction (`instr`), and another for implementing the security policy (Π). Under our instrumentation, a browser does not interpret a document D directly. Instead, it interprets a rewritten version $\iota(D)$ produced by the rewriting module. Upon encountering a special instruction `instr`(E), the implementation of `instr` evaluates the expression E and sends the result-

ing document D' through the rewriting module. The result of the rewriting $\iota(D')$ is directed back to the browser for further interpretation. Upon a call to the policy interface `check`(A), the policy module advances the state of the automaton and provides a replacement action A'.

In our prototype, the rewriting module is implemented in Java, with help of ANTLR [13] for parsing JavaScript code. We parse HTML documents into abstract syntax trees (ASTs), perform rewriting on the ASTs, and generate instrumented JavaScript code and HTML documents from the ASTs. We set up the browser to use this rewriting module as a proxy for all HTTP requests.

An obvious way to implement the special instruction is to modify the JavaScript interpreter in a browser according to the operational semantics given by Rule (33) in Section 5. After the interpreter parses a document piece out of the string argument (abstracted by the evaluation relation in Rule (33)), the rewriting module is invoked. The interpretation resumes afterwards with the rewritten document piece.

Although the above is straightforward, it requires changing the implementation of the browser. In our prototype, we opted for an implementation within the regular JavaScript language itself, where `instr` is implemented as a JavaScript function. The call-by-value nature of JavaScript functions evaluates the argument expression before executing the function body, which naturally provides the expected semantics. We make use of the `XMLHttpRequest` object [21] (popularly known as part of the Ajax [7] approach) to call the Java rewriting module from inside JavaScript code.

Although convenient, this approach is not as robust as that of modifying the JavaScript interpreter, because it is more vulnerable to malicious exploits. As discussed in Section 6, JavaScript provides some form of self-modifying code, *e.g.,* through `innerHTML`. This presents a possibility for malicious script to overwrite the implementation of `instr`, if `instr` is implemented in JavaScript and interpreted together with incoming documents. Additional code inspection is needed to protect against such exploits, which makes the implementation dependent on some idiosyncrasies of the JavaScript language. Therefore, it may be more desirable to modify the interpreter when facing a different tradeoff.

Similar implementation choices apply to the policy module. For example, one can implement the policy module as an add-on to the browser with the expected policy interface. In our prototype, we implemented the policy module also in regular JavaScript—`check` is implemented as a regular JavaScript function and calls to `check` are regular function calls with properly encoded arguments that reflect the actions being inspected. The body of the `check` function carries out the replacement actions, which are typically the original actions with checked arguments and/or inserted user prompts. The above protection concerns for `instr` against malicious exploits through self-modifying code also applies here.

We emphasize that our prototype enforces policies per "document cluster." A browser may simultaneously hold multiple windows. Some of these windows can communicate with each other (*e.g.,* a window and its pop-up, if the pop-up holds a document from the same origin); we consider these as being in the same cluster. We give each cluster its own policy instance in the form of a JavaScript object, and give all windows in the same cluster a reference to the cluster's policy instance, which is properly set up when windows are created or documents are loaded.

This does not affect the essence of the instrumentation, therefore we have elided its formal treatment (our CoreScript model only concerns a single cluster; our formal instrumentation only refers to a single policy instance). Nonetheless, the per-cluster enforcement is necessary for expressing practical policies. On the one hand, documents from different clusters should not share the same policy instance, so that the behavior of one document would not affect what an unrelated document is allowed to do (*e.g.,* two unrelated windows may each have their own quota of pop-ups). On the other hand, documents from the same cluster must share the same policy instance to prevent malicious exploits (*e.g.,* an attack may conduct relevant actions in separate documents in the same cluster).

8. Experiments

We implemented some simple but useful policies in the context of resource usage and secrecy, and tested the prototype with them on a number of web pages, including malicious or exploited pages from the real world, well-intended pages which might raise false positives, random popular pages, and some pages specially written to explore the boundaries of existing tools. Here we focus on two policies for demonstration purposes.

The first relates to controlling pop-up windows. Whereas many pop-up blockers have been developed and deployed, they only provide a limited degree of customization. Typically, the choices upon a pop-up are either to always allow, to allow once and ask again, or to disallow. In comparison, our tool is more flexible in the sense that it allows customization on the number of pop-ups allowed (see Figure 10) as well as the size, position and chrome visibility.

The second relates to controlling cookie information. Whereas browsers allow a webpage to access a cookie only if the cookie was set by the same domain, web pages with XSS vulnerabilities are still subject to cookie-stealing attacks (Section 2.1). We tried a simple policy which warns the user (and asks whether to proceed) if a webpage is sending network requests to a different domain after accessing the cookie (see Figure 11, where *safe-loadURI* is implemented to check domain conformance and prompt the user for decisions). We injected cookie-stealing script into some XSS-vulnerable web pages [14]. Our tool successfully raised warnings before the cookies were sent out. To the authors' knowledge, no existing tool provides client-side protection for such web pages.

We also tried the same cookie policy on some popular web pages for online banking, online shopping and web-based emails. We specifically looked for those that involved both cookie access and redirection to other domains, aiming to learn about false positives (in this case, these would be false warnings which users can easily dismiss). Only few tested pages exhibited policy-violating behaviors, and they were all from online shopping sites. The violation happened because (1) the cookie is accessed for login and/or shopping cart operations; (2) redirection to other domains happened when browsing external links, which are sometimes provided on product description pages. In contrast, none of the online banking or web-based email pages tested presented policy-violating behaviors. Some of them did not use cookies for storing login information at all. Others performed redirection before handling cookies. This, in retrospection, is reasonable if the cookie is used to store login information. Redirection after cookie access would happen if a website sets a cookie from one domain but handles login from another, a questionable behavior on its own.

We did not measure the performance of the prototype, because JavaScript additions to web pages are usually small and the instrumentation overhead is unlikely to be noticeable given the nature of web pages. Indeed, we did not notice any performance difference between the original and the instrumented web pages. For the same reason, our prototype does not perform optimizations, even though it is possible to avoid certain checks by some analysis on the code.

We point out that these preliminary experiments aim at demonstrating the effectiveness of our instrumentation method and its potential in the context of browser security. A thorough policy investigation and use-case study is outside the scope of this paper. Nonetheless, we are actively investigating policy issues and experimenting with the latest attacks. For example, a simple policy that

warns against all redirections would have likely identified the recent Yahoo Mail Worm attack [19] and therefore helped control its damage. For a realistic deployment, customized policies that are specifically based on different domain names (*e.g.,* black and white lists) would also be useful. These are supported by our prototype, but not discussed due to space constraints.

9. Conclusion

JavaScript is widely employed to enhance web pages with client-side computation. Unfortunately, as a popular (and perceptually silent) form of mobile code, JavaScript has also been much exploited by malicious parties to launch browser-based attacks. Whereas modern browsers and separate security tools provide certain basic protections, there are still many attacks at large. It is useful to have a common and extensible framework that regulates the behavior of untrusted JavaScript code from the perspective of the JavaScript language, rather than from that of specific attacks only.

This paper presents a provably correct method for instrumenting JavaScript code for browser security. It reflects the application and adaptation of a few interesting language theories and techniques. We characterize the relevant features of JavaScript in a core language and give it an operational semantics addressing its different execution model. Due to the presence of higher-order script, the conventional method of instrumentation cannot effectively identify and rewrite all relevant code. We resolve this issue by embedding "callbacks" in the instrumented code, so that further rewriting on runtime-generated script can be carried out on demand. We use edit automata to express security policies, and present some simple consistency criterion and combination method for policy management. A policy interface separates the policy implementation from the rewriting mechanism, facilitating policy extension, upgrade and customization.

This paper has focused on describing our approach and proving its correctness. It is organized in a generic context so that the ideas are applicable to related questions. Our experiments are carried out with the actual JavaScript language, where the prototype implementation also tackles a few language-specific issues such as the dynamic binding and rebinding of properties and methods. We plan to document the implementation aspects separately. Current experiments show much promise on the effectiveness of the instrumentation method, although further investigation is needed on the practical aspects of deployment, particularly in the area of policy design and customization.

Acknowledgments

We wish to thank Zhong Shao and the anonymous referees for their helpful comments.

References

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 429–452, Glasgow, UK, July 2005.

[2] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. 2005 ACM Conference on Programming Language Design and Implementation*, pages 305–314, Chicago, IL, June 2005.

[3] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell. Client-side defense against web-based identity theft. In *Proc. 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2004.

[4] ECMA International. ECMAScript language specification. Stardard ECMA-262, 3rd Edition, `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec. 1999.

[5] 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.

[6] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. 20th IEEE Symposium on Security and Privacy*, pages 32–47, Oakland, CA, May 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] R. Hansen. XSS cheat sheet. Appendix of OWASP 2.0 Guide, `http://ha.ckers.org/xss.html`, Apr. 2005.

[9] A. L. Hors, P. L. Hegaret, L. W. ad Gavin Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) level 3 core specification. W3C candidate recommendation, `http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107/`, Nov. 2003.

[10] 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.

[11] 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, Washington, DC, 2004.

[12] MozillaZine. XPCNativeWrapper. MozillaZine Knowledge Base, `http://kb.mozillazine.org/XPCNativeWrapper`, 2006.

[13] T. Parr *et al.*. ANTLR reference manual. Reference manual, `http://www.antlr.org/`, Jan. 2005.

[14] Point Blank Security. The XSS blacklists. `http://www.pointblanksecurity.com/xss/` and `http://www.pointblanksecurity.com/xss/xss2.php`, 2002–2005.

[15] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2):138–168, May 2002.

[16] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceeding of the IEEE*, 63(9):1278–1308, Sept. 1975.

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

[18] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, pages 372–382, Charleston, SC, Jan. 2006.

[19] Symantec Corp. JS.Yamanner@m. Symantec Security Response, `http://www.symantec.com/security_response/writeup.jsp?docid=2006-061211-4111-99`, June 2006.

[20] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. 14th European Symposium on Programming*, pages 408–422, Edinburgh, UK, Apr. 2005.

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

[22] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, 1993.

[23] D. Walker. A type system for expressive security policies. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 254–267, Boston, MA, 2000.

[24] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th USENIX Security Symposium*, Vancouver, B.C., Canada, July 2006.