July 3, 2007

RT0736
Computer Science; Security 8 pages

Research Report

Secure Behavior of Web Browsers to Prevent Information Leakages

Takaaki Tateishi, Naoshi Tabuchi

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Secure Behavior of Web Browsers to Prevent Information Leakages

Takaaki Tateishi Naoshi Tabuchi
IBM Research, Tokyo Research Laboratory
{tate,tabee}@jp.ibm.com

Abstract

Recently Web browsers are widely used as client-side application platforms beyond the traditional use of Web browsers. One of main reasons for such evolution of the browsers is a client-side JavaScript language that can execute programs embedded in a document. However, Web applications with client-side JavaScript programs have problems in leaking private information (such as cookie information) due to interactions between a browser and scripts embedded in a document. In order to prevent Web browsers from leaking information, we propose a calculus representing browser behavior with considering information flows. The proposed calculus can deal with script rewriting and higher-order functions. In addition, our calculus has a non-interference property depending on a security policy statically given by a user.

1 Introduction

Web browsers are widely used as application platforms beyond simply showing structured documents. One of main reasons for this evolution of browser usage is that a client-side JavaScript supported in Web browsers can execute programs embedded in documents. However the interactions between scripts and documents can cause security problems.

Information Leakage from Browsers

Client-side JavaScript programs can retrieve highly confidential user information, such as passwords stored in cookies. In addition, these programs can modify the documents themselves. By combining these two functions, we can embed the confidential information into a document and then send that information to an arbitrary server by exploiting the browser's default behavior. Let us consider the following document as an example, and suppose that the document was received from a Web server domainB.

```
<body><script>
document.write(
  '<img src="http://domainA/'
  + document.cookie + '" />');
</script></body>
```

The JavaScript program in this document generates the following document including the cookie information `cookie`.

```
<img src="http://domainA/cookie" />
```

A browser interprets the generated document, and then reads an image file from the URL specified in the document by sending a request containing the cookie information. Thus the cookie information is exposed to the server in domainA.

Preventing Information Leakage by Information Flow

Information flow control[8] is useful for detecting information leakages. The common technique of keeping track of information dependencies in traditional programs is to annotate values or variables with security labels. Noninterference, which means that changes in high security input data does not cause any change in low security output data, ensures the safety of information flows. In order to benefit from such secure information flows to prevent information leakage, we consider the security labels as levels of confidentiality, and check the security levels of data passed to security sensitive operations such as a function sending data outside from a program.

Information Flows in Browsers

In the Web browser a hierarchal document called a DOM instance is like a target program executed by the browser and it represents a program state. This means that the document can be modified by a program embedded in the document itself, similar to a self-modifying program. To introduce information flow control to the Web browser, we have to consider (1) how to annotate DOM nodes with security labels, (2) how to formalize a noninterference property for a DOM instance, and (3) how to deal with self-modifying features that affects information flows. At far as we know, there is no information flow control handling the above considerations.

In this paper we propose a calculus to represent the behavior of the Web browser considering the information flow control, and then prove that a noninterference property holds for the calculus. In the calculus a set of URLs represents a security label and is assigned to a DOM node. If we suppose that the confidentiality level of the cookie information is $B = \{local, domainB\}$ for the previous example, then $B[$`document.cookie`$]$ represents the fact that the cookie information can be observed by the browser (local) and from the server in domainB. Such a security label is propagated to all of the DOM nodes appended by the program as follows.

```
B[<img B[src=B["http://domainA/cookie"]] />]
```

To detect information leakage we can check if the string value "http://domainA/cookie" of the `src` attribute can be observed by the domain domainA, before the browser sends any request to the domain, checking the resulting security label of the string value. In the above example the browser should not send the request to domainA since it includes information that should be observed only by the domains specified by B and $\{\text{domainA}\} \not\subseteq B$ holds.

The proposed behavior model of browsers is also designed so as to restrict any information flows caused by control dependencies, called implicit flows[8], in addition to the direct dependencies. This is necessary to avoid information leakages through such implicit flows. Suppose that `x` is a public variable that can be read from any domain in the following JavaScript program.

```
var x = false;  // x is not confidential.
if (document.cookie == "xyz")
  x = true;
document.write(
  '<img src="http://domainA/'
  + x + '" />');
```

The information that the value of the cookie equals to "xyz" is leaked even though the request to domainA doesn't contain the value of the cookie. Using a binary search technique in the similar code an unauthorized party could indirectly obtain complete information, as described by Vogt et al.[9]. When confidential values are used in the conditions of conditional branches and loops, such indirect information leakages are common. In our behavior model we prevent indirect information leakages by monitoring that confidentiality levels of values affecting control flows never violate a pre-determined security policy.

Related Work

SLam calculus [6] proposed by Heintze and Riecke is a typed lambda calculus extended with security types for information flows. Abadi et al. proposed dependency core calculus (DCC) [1] that provides a common framework for type-based dependency analyses similar to that of SLam calculus.

For procedural program languages, one of the well known languages is JFlow[7] by Myers, which is a statically typed language based on Java. Pistoia et al. proposed an access control mechanism based on a combination of the stack inspection[5] and an information flow technique. Their semantics satisfies a noninterference property. Banerjee and Naumann also proved that a noninterference property[3] holds for their procedural language. In addition, Barthe and Rezk proposed a security-typed calculus for JVM and proved a noninterference property holds for their calculus[4].

All of these prior results are based on a traditional lambda calculus or procedural languages, but don't deal with the feature of higher-order program where values are evaluated as programs. In addition, our model monitors information flows at runtime, which is different from the techniques of the prior research, since they rely on static or dynamic analyses.

There are some browser implementations or extensions to prevent information leakages. Netscape Navigator 3 with JavaScript 1.1 provided us with a data tainting mechanism. Anupam and Mayer proposed a security framework [2] for browsers, and Vogt et al. implemented a combination of a data tainting and static analyses[9] in Firefox.

There is a formal model for browser behavior called CoreScript[10] proposed by Yu et al. With CoreScript they describe constraints on browser behavior in the edit automata and check if a browser's execution satisfies the constraints by using program instrumentation. Their model reflects the behavior of the `document.write()` function that append a text into the document and execute a script embedded in the text as our model also reflects the same behavior. However they mainly addressed problems of resource usage, for example when programs tries to open infinitely many browser windows. They didn't deal with information leakages directly.

Organization

The rest of this paper is organized as follows. In the next section we describe our approaches to security labels and a noninterference property for DOM instances. In Section 3 we define the syntax and the operational semantics of a calculus representing browser behavior and in Section 4 we apply the calculus to a simple example to show how to detect an information leak. In Section 5 we conclude this paper and consider some future work.

2 Our Approach

This section describes how to keep track of information flows in documents and how to formalize the noninterference property for documents.

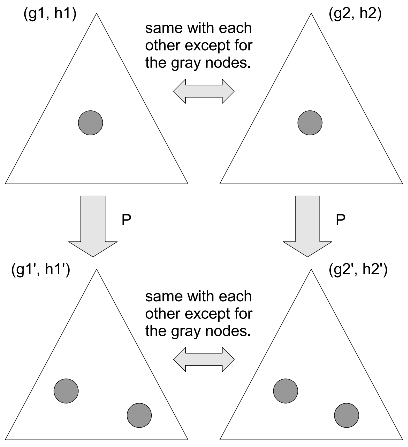
(g1, h1)
same with each other except for the gray nodes.
(g2, h2)
P
P
(g1', h1')
same with each other except for the gray nodes.
(g2', h2')

Figure 1. Noninterference for a DOM instance

2.1 Security Labels

In our information flow control, we annotate DOM nodes with security labels and each DOM node is represented by a tuple of g and h where g is a mapping@from parent nodes to child nodes, h is a mapping from nodes to tuples of a security label B and a node name v, and such a tuple (B, v) is denoted by $B[\![v]\!]$. For the example in Section 1, the DOM instance is formally represented as follows.

$$\begin{aligned} g &= \{(l_1, l_2), (l_2, l_3), (l_3, \varepsilon)\} \\ h &= \{(l_1, \mathcal{P}[\![\mathsf{body}]\!]), (l_2, \mathcal{P}[\![\mathsf{script}]\!]), \\ &\qquad (l_3, \mathcal{P}[\![\text{“}\mathtt{document.write(}\ldots\mathtt{)}\text{''}]\!])\} \end{aligned}$$

Intuitively the security label B represents a set of authorized observers who are allowed to obtain v and it would be represented by the domain of the document origin. A larger security label represents a lower confidentiality level since more observers can obtain that information. In addition the largest set of observers is denoted by $\mathcal{P}$. Thus highest and lowest confidentialities are represented by the security label $\emptyset$ and $\mathcal{P}$ respectively. The confidentiality level for an association from a parent node to a child node is defined to be the higher one. Thus such a security label is calculated by $R_p \cap R_c$.

2.2 Noninterference

A traditional noninterference property states that more confidential information is not affected by less confidential information. This property is formalized with two executions for a program, where $(P, \sigma) \Downarrow \sigma'$ represents the fact that if a program P executes from a program state σ, then the program terimnates in a state σ'.

$$\begin{aligned} &\sigma_1 \sim \sigma_2 \\ &\Rightarrow ((P, \sigma_1) \Downarrow \sigma'_1 \wedge (P, \sigma_2) \Downarrow \sigma'_2) \\ &\qquad \Rightarrow \sigma'_1 \sim \sigma'_2 \end{aligned}$$

The relation $\sim$ is called an indistinguishability, which defines that low-confidentiality values are the same as each other. [1] This definition indicates that high-confidentiality values are not comparable with each other since the values cannot be observed.

In accord with the above idea we formally define a noninterference property for DOM instances as follows

$$\begin{aligned} &(h_1, g_1) \sim (h_2, g_2) \\ &\Rightarrow ((P, (h_1, g_1)) \Downarrow (h'_1, g'_1) \wedge (P, (h_2, g_2)) \Downarrow (h'_2, g'_2)) \\ &\qquad \Rightarrow (h'_1, g'_1) \sim (h'_2, g'_2) \end{aligned}$$

where (h_1, g_1) and (h_2, g_2) are DOM instances and P is a program that modifies the DOM instances.

The relation $\sim$ in the above expression defines an equality (indistinguishability) for documents in which some portions cannot be compared with each other because of they are confidential. Figure 1 shows such a noninterference property in which the high security nodes, denoted by gray circles, are affecting two nodes and the low security nodes are the same as each other before and after the execution. Such indistinguishability characterizes the observational power of attackers and we formally define the indistinguishability in Section 3.

3 Calculus for Browser Behavior

A primary behavior of browsers is to handle embedded JavaScript programs. In this section, we model this browser behavior while considering information flows.

3.1 Syntax

We first define the syntax of our language used to represent browser behavior (as shown in Figure 2). The target which the browser interprets is a DOM instance. A DOM node is represented by its tag name t and an HTML attribute is represented by its attribute name with the prefix symbol @. A value v handled by the browser is a number literal n, a string literal str, a function f, or a boolean value b. $R[v]$ describes the value v with the security label R. A function with arguments $\vec{x}$ is denoted by $\mathsf{fun}(\vec{x})\{P; E\}$, where P is a body of the function and E is a return value. An expression E consists of variables x, values v, and operators op. The vectored $\vec{x}$ means a sequence of elements such as $x_1, x_2, \cdots, x_n$. A path expression pe is expressed by a variable x with postfix $.n$ where n is an integer value. In this path notation $x.n$ means the n-th child node of the node pointed by the variable x.

A program P is made up of common program constructs such as conditional branches, DOM operations, and check statements. In our model, we introduce three primitive DOM operations: write, append, and remove, corresponding to the JavaScript functions `document.write`,

[1]The equality of initial states is not always the same as the equality of final states. Refer to Section 3.A of the paper[8] written by Sabelfeld et al. for details.

$$
\begin{array}{llll}
(Variable) & x & \in & \mathcal{X} \\
(Number) & n & \in & \mathbf{N} \\
(Boolean) & b & \in & \mathbf{B} = \{\mathsf{true}, \mathsf{false}\} \\
(String) & str & \in & \Sigma* \\
(Tag) & t & \in & \mathbf{T} = \{\mathsf{script}, \mathsf{p}, \mathsf{@href}, \ldots\} \\
(Security) & R & \subseteq & \mathcal{P} \\
(Script) & P & ::= & \mathsf{skip} \\
 & & \mid & E \mid x = E \mid x = pe \mid P; P \\
 & & \mid & \mathsf{if}\ E\ \mathsf{then}\ P\ \mathsf{else}\ P \\
 & & \mid & \mathsf{check}\ R\ \mathsf{for}\ E \\
 & & \mid & \mathsf{write}(E) \mid \mathsf{append}(pe, pe) \\
 & & \mid & \mathsf{remove}(pe, pe) \\
(Expression) & E & ::= & x \mid v \mid op(\vec{E}) \mid E(E) \mid E(pe) \\
(PathExpression) & pe & ::= & x \mid pe.n \mid \mathsf{create}(E) \\
(Function) & f & ::= & \mathsf{fun}(\vec{x})\{P; E\} \\
(Value) & v & ::= & n \mid str \mid f \mid b \mid t \mid R[v] \\
(Operator) & op & ::= & + \mid - \mid * \mid /
\end{array}
$$

Figure 2. Syntax

`appendChild`, and `removeChild` respectively. A check statement checks the security label of a value. In addition, we drop any loop syntax from the definition, since we can use recursive function calls such as $f = \mathsf{fun}(\vec{x})\{\ldots; f(\vec{E}); \ldots\}$ instead of loops.

3.2 Semantics

The semantics of the language is defined in a big-step style with *a security policy* $\mathcal{Q} \subseteq \mathcal{P}$. In the semantics, a *program state* is denoted by (s, h, g). *A store* $s \in (\mathcal{X} \rightarrow L)$ is a partial mapping from variables $\mathcal{X}$ to *locations* (or addresses) L. *A heap* $h \in (L \rightarrow \mathcal{P} \times V)$ is a partial mapping from locations L to the tuples of security labels $\mathcal{P}$ and the values V where $V = \mathbf{N} \cup \Sigma * \cup \mathbf{B} \cup \mathbf{T} \cup (V \rightarrow V)$. A tuple such as $(R, v) \in 2^{\mathcal{P}} \times V$ is denoted by $R[\![v]\!]$, and we use the notation $R_1[\![R_2[\![v]\!]]\!]$ to express $R_1 \cap R_2[\![v]\!]$. *A DOM instance* $g \in (L \rightarrow \vec{L})$ is a partial mapping from locations to the sequences of locations.

In addition, we can annotate DOM nodes with different security labels, since each set of values contains tags T. This means that we can annotate a password input field `<input type=password .../>` with a higher confidentiality label than other nodes. We are sure that it is not so easy to appropriately configure security labels for DOM nodes in practical applications, since we have to consider how server-side programs deal with security labels and they should be consistent with client-side programs. However we suppose that security labels for DOM nodes are appropriately configured, and that issue is beyond the scope of this paper.

A sequence of locations is denoted by $(l_1, \cdots, l_n)$ and the empty sequence is denoted by ε. For convenience, we eliminate the parentheses for a sequence and flatten a sequence of sequences when it doesn't conflicts with other notations. For example, $(\vec{l})$ means $(l_1, \cdots, l_n)$ where $\vec{l} = (l_1, \cdots, l_n)$, and $(\vec{l}, \varepsilon, \vec{l})$ means $(l_1, \cdots, l_n, l_1, \cdots, l_n)$. In addition, undefined value is denoted by $\perp$ for the partial mappings.

The policy $\mathcal{Q}$ intuitively indicates a set of trusted domains. [2] We require the policy to prohibit implicit flows that possibly cause information leakages to untrusted domains, which are not included in $\mathcal{Q}$. This is because we need a static analysis technique to investigate all of execution paths[8, 9] to track the implicit flows if we don't use a static policy. However such a static analysis is not practical, since our calculus allows program modifications at runtime.

Finding Scripts

The browser constructs a DOM instance by interpreting an HTML document. We represents that operation with $\mathsf{parse}^{\mathsf{D}}$. $\mathsf{parse}^{\mathsf{D}}(v, h, g, R)$ returns a triple (l, h', g') where l is a location pointing to the root node of a constructed DOM instance represented by the tuple (h', g'). In addition, every created location l and the corresponding value v representing tags or strings are supposed to satisfy the equation $h'(l) = R[\![v]\!]$. Suppose that $(s_{init}, h_{init}, g_{init})$ represents a browser's state after the DOM construction, and the variable doc points to the root node of the DOM instance, then the following formal statement represents the fact that the browser executes embedded JavaScript programs and terminates at the final state (s', h', g').

$$(s_{init}(\mathsf{doc}), s_{init}, h_{init}, g_{init}) \Downarrow (s', h', g')$$

The evaluation rules for calculating the final state are described in Figure 3. The rules of X-NODE and X-LEAF represent traversing a DOM instance in depth first order, and the rule of X-SCRIPT represents script executions. In the rules the variable current and parent point to a currently visited node and the parent node of the current node respectively. $P = \mathsf{parse}^{\mathsf{S}}(str)$ in the rule of X-SCRIPT represents the fact that a program P is obtained by parsing the string str. In addition, we define the rules of X-NODE and X-SCRIPT so as to traverse each node only when the security label is greater than the policy $\mathcal{Q}$ to prevent occurrences of implicit flows caused by dynamically updated DOM nodes.

Script Execution

The semantics of the script executions consists of rules for expressions, scripts and DOM operations as described in

[2]In general, the policy $\mathcal{Q}$ and the security labels $R \subseteq \mathcal{P}$ might be an infinite set of domains. However, it is required here that the judgment of $R_1 \subseteq R_2$ should terminates in finite steps for all security labels R_1 and R_2. From a practical perspective, we could use regular expressions to denote a set of domains such as $\mathcal{Q} = \{$`http://[a-z]+.ibm.com`$\}$.

$$\frac{\begin{array}{c} h(l) = R[\![v]\!] \quad v \neq \mathsf{script} \quad g(l) = (l_1, l_2, \cdots, l_n) \quad h(l_i) = R_i[\![v_i]\!] \quad 1 \leq i \leq n \quad s_0 = s \quad h_0 = h \quad g_0 = g \\ \begin{cases} (l_i, s_{(i-1)}[\mathsf{parent} \mapsto l, \mathsf{current} \mapsto l_i], h_{(i-1)}, g_{(i-1)}) \Downarrow (s_i, h_i, g_i) & R \cap R_i \supseteq \mathcal{Q} \\ (s_{(i-1)}, h_{(i-1)}, g_{(i-1)}) = (s_i, h_i, g_i) & \text{(otherwise)} \end{cases} \end{array}}{(l, s, h, g) \Downarrow (s_n, h_n, g_n)} \text{ X-NODE}$$

$$\frac{\begin{array}{c} h(l) = R[\![\mathsf{script}]\!] \quad R \supseteq \mathcal{Q} \quad g(l) = l_s \quad h(l_s) = R_s[\![str]\!] \quad R_s \supseteq \mathcal{Q} \\ P = \mathsf{parse}^{\mathsf{S}}(str) \quad (P, s, h, g) \Downarrow (s', h', g') \end{array}}{(l, s, h, g) \Downarrow (s', h', g')} \text{ X-SCRIPT} \qquad \frac{g(l) = \varepsilon}{(l, s, h, g) \Downarrow (s, h, g)} \text{ X-LEAF}$$

Figure 3. Rules for finding embedded scripts

Figure 4, 5 and 6 respectively. The rules for expressions are defined so as to implement call-by-value and left-to-right evaluation.

$(E, s, h, g) \Downarrow (v, s', h', g')$ means that the expression E in the state (s, h, g) evaluates to the value $v \in L \cup (2^{\mathcal{P}} \times V)$ and terminates in the state (s', h', g'). As for rules for scripts and DOM operations, we use the form $(P, s, h, g) \Downarrow (s', h', g')$. In addition, "$l$ is fresh" means that l is a newly generated location.

In the rule of P-APPEND in Figure 6, when a DOM node is appended to a DOM tree and its root node is doc, the browser finds scripts in the tree. This behavior is formally defined by the function $\mathcal{E}$ where $reachable(l, l', g)$ is a predicate that means that there exists a path from l to l'. For example, the following script appends a script to a DOM node, and then the appended script is evaluated.

$sn = \mathsf{create}(\mathsf{script});$
$sb = \mathsf{create}(\texttt{"script code"})$
$\mathsf{append}(sn, sb);$
$\mathsf{append}(\mathsf{doc}.0, sn)$

Notice that if we remove the last command $\mathsf{append}(\mathsf{doc}.0, sn)$, the generated script is not evaluated until it is appended to the subtree of doc.

The rules of P-IF, E-APP, and EL-APP is defined so as to apply if and only if the security label R for a condition value or an application function is greater than or equal to the policy $\mathcal{Q}$. This is because we prohibit implicit flows caused by variations of condition values and application functions.

3.3 Noninterference

In this section we show how to prove the noninterference property [8, 3] that ensures that data of higher security levels than Q don't affect the lower security levels. For this purpose, we first define which information can be retrieved by observers who are allowed to access data of security levels that are lower than or equal to Q. We call such observers Q-observers.

Definition 1

$$\begin{array}{l} Low_Q(s, h, g) \equiv (s', h', g') \\ \textit{where} \\ s' = \{(x, l) \mid l = s(x),\ R[\![v]\!] = h(l),\ R \supseteq Q\} \\ h' = \{(l, v) \mid R[\![v]\!] = h(l),\ R \supseteq Q\} \\ g' = \{(l_p, (l_{c_1}, \cdots, l_{c_k})) \mid \\ \qquad g(l_p) = l_1, \cdots, l_{c_1}, \cdots, l_{c_k}, \cdots, l_n, \\ \qquad h(l_p) = R_p[\![v_p]\!],\ h(l_{c_i}) = R_{c_i}[\![v_{c_i}]\!],\ 0 \leq i \leq k, \\ \qquad R_p \cap R_{c_i} \supseteq Q\} \end{array}$$

In this definition, s' is a relation between variables and values, h' is a relation between DOM nodes and its names and g' is a parent-children relation including a sibling relation of DOM nodes.

In general, if the equation $Low_Q(s_1, h_1, g_1) = Low_Q(s_2, h_2, g_2)$ holds, it is natural to use Low_Q to express the noninterference property, since the equation means that Q-observers cannot distinguish (s_1, h_1, g_1) from (s_2, h_2, g_2). However, fresh locations are generated nondeterministically. This means that locations newly generated at the same state are not always the same. Thus we define the following equality $=_\beta$ on locations using a bijective correspondence $\beta \in (L \rightarrow L)$ between those locations, as described by Barthe et al.[4].

Definition 2

$$\begin{array}{l} (s_1, h_1, g_1) =_\beta (s_2, h_2, g_2) \\ \equiv \mathsf{dom}(s_1) = \mathsf{dom}(s_2) \wedge \beta(\mathsf{dom}(h_1)) = \mathsf{dom}(h_2) \\ \quad \wedge (\forall x \in \mathsf{dom}(s_1)\ .\ \beta(s_1(x)) = s_2(x)) \\ \quad \wedge (\forall l \in \mathsf{dom}(h_1)\ .\ h_1(l) = h_2(\beta(l)) \\ \qquad\qquad \wedge \beta(g_1(l)) = g_2(\beta(l))) \end{array}$$

In this definition, we extend the definition of β to the set of locations and to the sequence of locations for convenience: $\beta(\mathsf{dom}(\mathsf{h}_1))$ denotes $\{\beta(h_1(l)) | l \in \mathsf{dom}(h_1)\}$. Intuitively, $(s_1, h_1, g_1) =_\beta (s_2, h_2, g_2)$ means that two states are equivalent if the memory layout of the data is ignored.

Next we define the indistinguishability $\sim_Q^\beta$ using Low_Q and $=_\beta$ as follows.

$$\frac{v \in \mathit{Value}}{(v, s, h, g) \Downarrow (\mathcal{P}\llbracket v \rrbracket, s, h, g)}\ \text{E-VAL} \quad \frac{v \in \mathit{Value}}{(R[v], s, h, g) \Downarrow (R\llbracket v \rrbracket, s, h, g)}\ \text{E-SVAL} \quad \frac{\begin{array}{c} l = s(x) \quad l \notin \mathsf{dom}(g) \\ R\llbracket v \rrbracket = h(l) \end{array}}{(x, s, h, g) \Downarrow (R\llbracket v \rrbracket, s, h, g)}\ \text{E-VAR}$$

$$\frac{(pe, s, h, g) \Downarrow (l_p, s', h', g') \quad l_1, \cdots, l_n, \cdots, l_m = g(l_p)}{(pe.n, s, h, g) \Downarrow (l_n, s', h', g')}\ \text{E-LPATH} \quad \frac{l = s(x) \quad l \in \mathsf{dom}(g)}{(x, s, h, g) \Downarrow (l, s, h, g)}\ \text{E-LVAR}$$

$$\frac{(E, s, h, g) \Downarrow (R\llbracket v \rrbracket, s', h', g') \quad l \text{ is fresh} \quad h'' = h'[l \mapsto R\llbracket v \rrbracket] \quad g'' = g'[l \mapsto \varepsilon]}{(\mathsf{create}(E), s, h, g) \Downarrow (l, s', h'', g'')}\ \text{E-CREATE}$$

$$\frac{\begin{array}{c} op \in \{+, -, *, /\} \quad (E_1, s, h, g) \Downarrow (R_1\llbracket v_1 \rrbracket, s_1, h_1, g_1) \quad (E_2, s_1, h_1, g_1) \Downarrow (R_2\llbracket v_2 \rrbracket, s_2, h_2, g_2) \\ v \text{ is the result of corresponding numeric/string operation on } v_1 \text{ and } v_2 \end{array}}{(op(E_1, E_2), s, h, g) \Downarrow (R_1 \cap R_2\llbracket v \rrbracket, s_2, h_2, g_2)}\ \text{E-OPS-NUM}$$

$$\frac{\begin{array}{c} (E_f, s, h, g) \Downarrow (R_f\llbracket f \rrbracket, s_0, h_0, g_0) \quad R_f \supseteq \mathcal{Q} \quad f = \mathsf{fun}(x)\{P; E_r\} \quad x \notin \mathsf{dom}(s) \\ (E, s_0, h_0, g_0) \Downarrow (R_v\llbracket v \rrbracket, s_e, h_e, g_e) \quad l \text{ is fresh} \quad s'_e = s_e[x \mapsto l] \quad h'_e = h_e[l \mapsto R_v\llbracket v \rrbracket] \\ (P, s'_e, h'_e, g_e) \Downarrow (s_f, h_f, g_f) \quad (E_r, s_f, h_f, g_f) \Downarrow (R_r\llbracket v_r \rrbracket, s_r, h_r, g_r) \end{array}}{(E_f(E), s, h, g) \Downarrow (R_f \cap R_r\llbracket v_r \rrbracket, s_r, h_r, g_r)}\ \text{E-APP}$$

$$\frac{\begin{array}{c} (E_f, s, h, g) \Downarrow (R_f\llbracket f \rrbracket, s_0, h_0, g_0) \quad R_f \supseteq \mathcal{Q} \quad f = \mathsf{fun}(x)\{P; E_r\} \quad x \notin \mathsf{dom}(s) \\ (pe, s_0, h_0, g_0) \Downarrow (l, s_e, h_e, g_e) \quad s'_e = s_e[x \mapsto l] \\ (P, s'_e, h_e, g_e) \Downarrow (s_f, h_f, g_f) \quad (E_r, s_f, h_f, g_f) \Downarrow (R_r\llbracket v_r \rrbracket, s_r, h_r, g_r) \end{array}}{(E_f(pe), s, h, g) \Downarrow (R_f \cap R_r\llbracket v_r \rrbracket, s_r, h_r, g_r)}\ \text{EL-APP}$$

Figure 4. Semantics for expressions

$$\frac{(E, s, h, g) \Downarrow (R\llbracket v' \rrbracket, s', h', g')}{(E, s, h, g) \Downarrow (s', h', g')}\ \text{P-EXP} \quad \frac{(E, s, h, g) \Downarrow (R_v\llbracket v \rrbracket, s', h', g') \quad R \subseteq R_v}{(\mathsf{check}\ R\ \mathsf{for}\ E, s, h, g) \Downarrow (s', h', g')}\ \text{P-CHECK}$$

$$\frac{\begin{array}{c} (E, s, h, g) \Downarrow (R\llbracket v \rrbracket, s', h', g') \\ l \text{ is fresh} \quad s'' = s'[x \mapsto l] \quad h'' = h'[l \mapsto R\llbracket v \rrbracket] \end{array}}{(x = E, s, h, g) \Downarrow (s'', h'', g')}\ \text{P-ASN} \quad \frac{(pe, s, h, g) \Downarrow (l, s', h', g') \quad s'' = s'[x \mapsto l]}{(x = pe, s, h, g) \Downarrow (s'', h', g')}\ \text{P-LASN}$$

$$\frac{}{(\mathsf{skip}, s, h, g) \Downarrow (s, h, g)}\ \text{P-SKIP} \quad \frac{(P_1, s, h, g) \Downarrow (s', h', g') \quad (P_2, s', h', g') \Downarrow (s'', h'', g'')}{(P_1; P_2, s, h, g) \Downarrow (s'', h'', g'')}\ \text{P-SEQ}$$

$$\frac{\begin{array}{c} (E, s, h, g) \Downarrow (R\llbracket v \rrbracket, s_e, h_e, g_e) \quad R \supseteq \mathcal{Q} \\ P' = \left\{ \begin{array}{ll} P_2 & (v = \mathsf{false} \lor v = 0) \\ P_1 & (\text{otherwise}) \end{array} \right. \quad (P', s_e, h_e, g_e) \Downarrow (s', h', g') \end{array}}{(\mathsf{if}\ E\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, s, h, g) \Downarrow (s', h', g')}\ \text{P-IF}$$

Figure 5. Semantics for scripts

$$\dfrac{\begin{array}{c}(E,s,h,g) \Downarrow (R[\![v]\!], s_e, h_e, g_e) \quad R \supseteq \mathcal{Q} \quad v \in \Sigma* \quad (l, h_d, g_d) = \mathsf{parse}^{\mathsf{D}}(v, h_e, g_e, R) \\ \vec{l_1}, s_e(\mathsf{current}), \vec{l_2} = g(s_e(\mathsf{parent})) \quad g'_d = g_d[s_e(\mathsf{parent}) \mapsto \vec{l_1}, s_e(\mathsf{current}), l, \vec{l_2}] \\ (l, s_e, h_d, g'_d) \Downarrow (s_n, h_n, g_n)\end{array}}{(\mathsf{write}(E), s, h, g) \Downarrow (s_n, h_n, g_n)} \text{ P-WRITE}$$

$$\dfrac{\begin{array}{c}(pe_p, s, h, g) \Downarrow (l_p, s_p, h_p, g_p) \quad (pe_c, s_p, h_p, g_p) \Downarrow (l_c, s_c, h_c, g_c) \\ \vec{l} = g_c(l_p) \quad g' = g_c[l_p \mapsto \vec{l}, l_c] \\ s_n, h_n, g_n = \mathcal{E}(l_c, s_c, h'_c, g')\end{array}}{(\mathsf{append}(pe_p, pe_c), s, h, g) \Downarrow (s_n, h_n, g_n)} \text{ P-APPEND}$$

$$\dfrac{\begin{array}{c}(pe_p, s, h, g) \Downarrow (l_p, s_p, h_p, g_p) \quad (pe_c, s_p, h_p, g_p) \Downarrow (l_c, s_c, h_c, g_c) \\ \vec{l_{c1}}, l_c, \vec{l_{c2}} = g_c(l_p) \quad g' = g_c[l_p \mapsto \vec{l_{c1}}, \vec{l_{c2}}]\end{array}}{(\mathsf{remove}(pe_p, pe_c), s, h, g) \Downarrow (s_c, h', g')} \text{ P-REMOVE}$$

$$\mathcal{E}(l, s, h, g) \equiv \begin{cases} (s', h', g') & \mathit{reachable}(s(\mathsf{doc}), l, g) \wedge (l, s, h, g) \Downarrow (s', h', g') \\ (s, h, g) & \text{(otherwise)} \end{cases}$$

Figure 6. Semantics for the DOM operations

Definition 3 (Q, β-indistinguishability)

$$\begin{aligned}&(s_1, h_1, g_1) \sim_Q^\beta (s_2, h_2, g_2) \\ &\quad \equiv \mathit{Low}_Q(s_1, h_1, g_1) =_\beta \mathit{Low}_Q(s_2, h_2, g_2)\end{aligned}$$

Here, β is used for the correspondence of locations, and Q is used for classifying security labels into low and high security classes.

The noninterference property for the browser behavior is as follows where $\mathcal{Q} \supseteq Q$ must hold for the static policy $\mathcal{Q}$.

Theorem 1 (Q-noninterference)

$$\begin{aligned}&\forall \beta, s_1, h_1, g_1, s_2, h_2, g_2 \,.\, (s_1, h_1, g_1) \sim_Q^\beta (s_2, h_2, g_2) \\ &\quad \Rightarrow \exists s'_1, h'_1, g'_1, s'_2, h'_2, g'_2 \,.\, \exists \beta' \supseteq \beta \,. \\ &\qquad ((s_1(\mathsf{doc}), s_1, h_1, g_1) \Downarrow (s'_1, h'_1, g'_1) \\ &\qquad \wedge (s_2(\mathsf{doc}), s_2, h_2, g_2) \Downarrow (s'_2, h'_2, g'_2)) \\ &\qquad \Rightarrow (s'_1, h'_1, g'_1) \sim_Q^{\beta'} (s'_2, h'_2, g'_2)\end{aligned}$$

In addition, $\exists \beta' \supseteq \beta$ means that there exists a location mapping β' that contains a mapping for freshly generated locations and preserves a location mapping β. We outline the proof of this theorem in Appendix A.

4 Case Study

Figure 7 shows how to apply the calculus proposed in Section 3 to the example from Section 1, where we use B as the static policy and we omit trivial conditions.

The triple $(s_{init}, h_{init}, g_{init})$ is an initial state constructed by the browser when interpreting a document, and thus $(l_1, s_{init}, h_{init}, g_{init}) \Downarrow (s_2, h_3, g_3)$ means that the browser traveses and executes the DOM instance and terminates at a final state (s_2, h_3, g_3). The relationship between the initial state and the final state is derived from the evaluation relationship $(l_2, s_2, h_{init}, g_{init}) \Downarrow (s_2, h_3, g_3)$ by applying the X-NODE rule. In the same way, we can derive the evaluation relationship from the other relationships by applying appropriate rules.

Finally the browser tries to get an image file from the domain domainA since the DOM instance obtained after the execution is (h_3, g_3). At this time we can prevent a leakage of confidential information by checking if the security label of $B[\![$"http://domainA/cookie"$]\!]$ satisfies $\{domainA\} \not\subseteq B$ using the check statement. In addition, from the B-noninterference property, we can ensure that the values of the security label $B' = \{domainA, domainB, local\}$, which could possibly leak to domainA, don't affect the values of the security label B.

5 Conclusion and Future Work

In this paper we proposed a calculus representing browser behavior. The calculus is defined to consider information flow control so as to express interactive behaviors between documents and embedded scripts. It also satisfies the noninterference property depending on the static security policy $\mathcal{Q}$.

In addition, our semantics requires a restriction, under which values varying control flows have to be less confidential than the security policy $\mathcal{Q}$, in order to prohibit implicit flows. We are aware that this restriction on the semantics is not practical. To relax the restriction is one of the challenges for the future. The other future goal is to model an access control for integrity, since we think that some kinds of security problems like cookie falsification can be prevented by access control.

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{(l_4, s_2, h_3, g_3) \Downarrow (s_2, h_3, g_3)}}{(\mathsf{write}(\ldots), s_{init}, h_{init}, g_{init}) \Downarrow (s_2, h_3, g_3)}\ \text{P-WRITE}}{(l_2, s_2, h_{init}, g_{init}) \Downarrow (s_2, h_3, g_3)}\ \text{X-SCRIPT}}{(l_1, s_{init}, h_{init}, g_{init}) \Downarrow (s_2, h_3, g_3)}\ \text{X-NODE}$$

$$\begin{array}{rcl}
s_{init} & = & [\mathsf{doc} \mapsto l_1, \mathsf{cookie} \mapsto B[\![\text{"cookie"}]\!]] \\
h_{init} & = & [l_1 \mapsto \mathcal{P}[\![\mathsf{body}]\!], l_2 \mapsto \mathcal{P}[\![\mathsf{script}]\!], \\
 & & \quad l_3 \mapsto \mathcal{P}[\![\text{“}\texttt{document.write}(\ldots)''\,]\!]] \\
g_{init} & = & [l_1 \mapsto l_2, l_2 \mapsto l_3, l_3 \mapsto \varepsilon] \\
s_2 & = & s_{init}[\mathsf{parent} \mapsto l_1, \mathsf{current} \mapsto l_2] \\
h_3 & = & h_{init}[l_4 \mapsto B[\![\mathsf{img}]\!], l_5 \mapsto B[\![\mathsf{@src}]\!], \\
 & & \quad l_6 \mapsto B[\![\text{"http://domainA/cookie"}]\!]] \\
g_3 & = & g_{init}[l_1 \mapsto (l_2, l_4), l_4 \mapsto l_5, l_5 \mapsto l_6, l_6 \mapsto \varepsilon]
\end{array}$$

Figure 7. Example of an execution

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] V. Anupam and A. Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *the 7th USENIX Security Symposium*, pages 187–200, 1998.

[3] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[4] G. Barthe and T. Rezk. Non-interference for a jvm-like language. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[5] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[6] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[7] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[8] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1), 2003.*, 2003.

[9] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[10] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM Press.

A Proof of Theorem 1

The proof is derived from the following lemma that can be proved by induction on $\Downarrow$ where all the rules for scripts and DOM operations are translated into the form for expressions. For example, we consider $(P, s, h, g) \Downarrow (s', h', g')$ as $(P, s, h, g) \Downarrow (\mathcal{P}[\![0]\!], s', h', g')$

Lemma 1

$$\begin{array}{l}
\forall \beta, s_1, h_1, g_1, s_2, h_2, g_2 \,.\, (s_1, h_1, g_1) \sim_Q^\beta (s_2, h_2, g_2) \\
\quad \Rightarrow \exists s_1', h_1', g_1', s_2', h_2', g_2' \,.\, \exists \beta' \supseteq \beta \,. \\
\qquad ((E, s_1, h_1, g_1) \Downarrow (R_1[\![v_1']\!], s_1', h_1', g_1') \\
\qquad \wedge (E, s_2, h_2, g_2) \Downarrow (R_2[\![v_2']\!], s_2', h_2', g_2')) \\
\qquad \Rightarrow (s_1', h_1', g_1') \sim_Q^{\beta'} (s_2', h_2', g_2') \\
\qquad\quad \wedge (R_1 \supseteq Q \wedge R_2 \supseteq Q \Rightarrow v_1' = v_2')
\end{array}$$

The proof of the induction step for the P-IF rule is as follows: Let us consider two states (s_1, h_1, g_1), (s_2, h_2, g_2) and the IF statement $P_{if} = \mathsf{if}\ E\ \mathsf{then}\ P_a\ \mathsf{else}\ P_b$. The following formula holds from the induction hypothesis.

$$\begin{array}{l}
(s_1, h_1, g_1) \sim_Q^\beta (s_2, h_2, g_2) \\
\Rightarrow ((E, s_1, h_1, g_1) \Downarrow (R_1[\![v_1]\!], s_{c1}, h_{c1}, g_{c1}) \wedge \\
\quad (E, s_2, h_2, g_2) \Downarrow (R_2[\![v_2]\!], s_{c2}, h_{c2}, g_{c2}) \\
\quad \Rightarrow \exists \beta' \supseteq \beta \,.\, ((s_{c1}, h_{c1}, g_{c1}) \sim_Q^\beta (s_{c2}, h_{c2}, g_{c2})) \wedge \\
\qquad \wedge (R_1 \supseteq Q \wedge R_2 \supseteq Q \Rightarrow v_1 = v_2)
\end{array}$$

In addition, $v_1 = v_2$ holds because of $R_1 \supseteq \mathcal{Q} \supseteq Q$ and $R_2 \supseteq \mathcal{Q} \supseteq Q$. Thus the same program (P_a or P_b) is executed. If P_a is executed, the following formula holds from the induction hypothesis.

$$\begin{array}{l}
(s_{c1}, h_{c1}, g_{c1}) \sim_Q^{\beta'} (s_{c2}, h_{c2}, g_{c2}) \\
\Rightarrow ((P_a, s_{c1}, h_{c1}, g_{c1}) \Downarrow (\mathcal{P}[\![0]\!], s_1', h_1', g_1') \\
\quad (P_a, s_{c2}, h_{c2}, g_{c2}) \Downarrow (\mathcal{P}[\![0]\!], s_2', h_2', g_2')) \\
\quad \Rightarrow \exists \beta'' \supseteq \beta' \,.\, ((s_1', h_1', g_1') \sim_Q^{\beta''} (s_2', h_2', g_2'))
\end{array}$$

Therefore, the following formula holds.

$$\begin{array}{l}
(s_1, h_1, g_1) \sim_Q^\beta (s_2, h_2, g_2) \\
\Rightarrow ((P_{if}, s_1, h_1, g_1) \Downarrow (\mathcal{P}[\![0]\!], s_1', h_1', g_1') \\
\quad (P_{if}, s_2, h_2, g_2) \Downarrow (\mathcal{P}[\![0]\!], s_2', h_2', g_2')) \\
\quad \Rightarrow \exists \beta'' \supseteq \beta \,.\, ((s_1', h_1', g_1') \sim_Q^{\beta''} (s_2', h_2', g_2'))
\end{array}$$

We can prove the case for P_b in the same way.