An Information Flow Monitor for a Core of DOM

Ana Almeida Matos, José Fragoso Santos, Tamara Rezk

An Information Flow Monitor for a Core of DOM

Introducing references and live primitives

Ana Almeida Matos[1,2], José Fragoso Santos[3], and Tamara Rezk[3]

[1] SQIG–Instituto de Telecomunicações, Lisbon, Portugal
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
[3] Inria, Sophia Antipolis, France

Abstract. We propose and prove sound a novel, purely dynamic, flow-sensitive monitor for securing information flow in an imperative language extended with DOM-like tree operations, that we call Core DOM. In Core DOM, as in the DOM API, tree nodes are treated as first-class values. We take advantage of this feature in order to implement an information flow control mechanism that is finer-grained than previous approaches in the literature. Furthermore, we extend Core DOM with additional constructs to model the behavior of *live collections* in the DOM Core Level 1 API. We show that this kind of construct effectively augments the observational power of an attacker and we modify the proposed monitor so as to tackle newly introduced forms of information leaks.

1 Introduction

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [11]. In contrast to the ECMA Standard [1] that specifies in full detail the internals of objects created during the execution, the DOM API only specifies the behavior that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine but by a separate engine whose role is to do so. Therefore, the design of an information flow monitor for client-side JavaScript Web applications must take into account the DOM API.

Russo et al. [13] first studied the problem of information flow control in dynamic tree structures, for a model where programs are assumed to operate on a single current working node. However, in real client-side JavaScript, tree nodes are first-class values, which means that a program can store in memory several references to different nodes in the DOM forest at the same time. We present a flow-sensitive monitor for tracking information flow in a DOM-like language, that we call Core DOM. In Core DOM, tree nodes are treated as first-class values and thus they support all operations available to other types of values, such as assignment to variables. Interestingly, this language design feature enables us to implement a more fine-grained information flow control mechanism, since it becomes possible to distinguish the security level of the node itself from both the security level of the value that is stored in the node and from the level of its

position in the DOM forest. We prove that the proposed monitor is sound with respect to a standard definition of noninterference.

Live collections are a special kind of data structure featured in the DOM API that automatically reflects the changes that occur in the document. There are several types of live collections. For instance, the method `getElementsByTagName` returns a live collection containing the DOM nodes that match a given *tag*. In the following example, after retrieving the initial collection of DIV nodes, the program iterates over the *current* size of this collection, while introducing a new DIV node at each step:

```
divs = document.getElementsByTagName("DIV"); i = 0;
while(i <= divs.length){
  document.appendChild(document.createElement("DIV")); i++; }
```

Every time a new DIV node is inserted in the `document` (no matter where in its structure), it is also inserted in the live collection bound to `divs`. Due to the live update of the loop condition, if the initial `document` contains at least one DIV node, the program does not terminate.

Live collections can be exploited to encode new types of information leaks. Therefore, we extend Core DOM with two additional constructs that model the behavior of `getElementsByTagName` in the DOM API. We demonstrate that these constructs effectively augment the observational power of an attacker and we show how to modify the proposed monitor so as to tackle this issue.

In the remainder of the paper, we start by formally introducing the target language Core DOM (Section 2). We then present a mechanism that controls information flows by means of a flow-sensitive monitor (Section 3). The scenario is then extended with live collections, for which we propose a modified mechanism (Section 4). Finally, we discuss related work and conclude. Due to space restrictions, proofs are presented in the article's full version [2].

2 Core DOM

We now present Core DOM – a simple imperative language extended with primitives for operating on tree structures. Its syntax is given by the following:

$e ::= v$	literal value	$\mid \underline{v}$	runtime value
$\mid x$	identifier	$\mid \mathsf{if}(e_0)\{e_1\}\ \mathsf{else}\ \{e_2\}$	conditional
$\mid \mathsf{while}(e_0)\{e_1\}$	loop	$\mid \mathsf{while}^{e_0}(e_1)\{e_2\}$	internal loop
$\mid e_0; e_1$	sequence	$\mid \mathsf{move}_{\uparrow}(e)$	move upward
$\mid \mathsf{move}_{\downarrow}(e, e)$	move downward	$\mid \mathsf{remove}(e, e)$	remove node
$\mid \mathsf{insert}(e, e, e)$	insert node	$\mid \mathsf{new}^{\sigma_0,\sigma_1,\sigma_2}(e)$	new node
$\mid \mathsf{value}(e)$	node value	$\mid \mathsf{store}(e, e)$	store value
$\mid \mathsf{len}(e)$	node length	$\mid \mathsf{end}(e)$	end

Every tree node has a type, called its *tag* (for instance, DIV) and can store a single value taken from a set $\mathcal{P}rim$ containing integers, strings, booleans, and a special value *null*. All the nodes in memory form a *forest*, meaning that every node has a possibly empty list of *children* and at most a single *parent*. We define the *index* of a node as the position it occupies in the list of children of its parent.

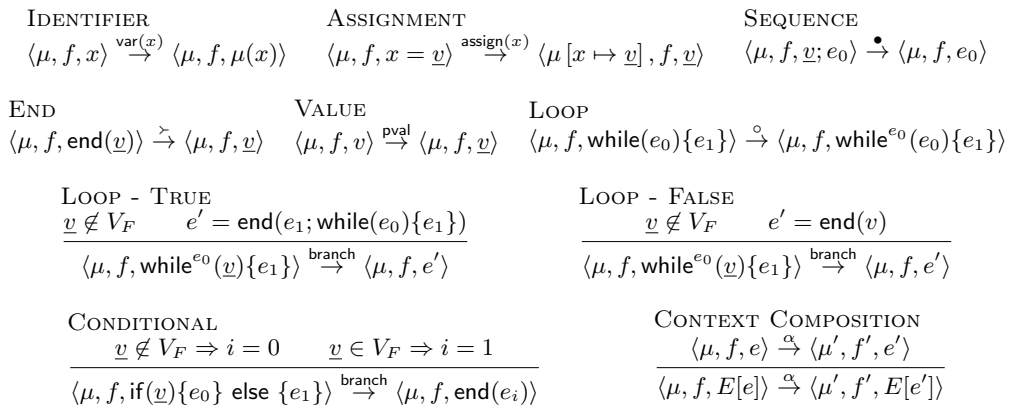


Fig. 1. Core DOM Semantics - Imperative Fragment

New nodes are created using the primitive new, which expects as input the tag of the node to be created and is annotated with three security levels that are explained later. When given a node as input, the primitive $\mathsf{move}_{\uparrow}$ evaluates to its parent in the DOM forest. Complementarily, the primitive $\mathsf{move}_{\downarrow}$ expects as input a node n and an integer i and evaluates to the i^{th} child of n. The primitive insert is used for inserting an *orphan node* (that is, a node with no parent) in the list of children of another node, whereas the primitive remove is used for removing a node from the list of children of its parent. Concretely, insert expects as input two nodes n_0 and n_1 and an integer i and inserts n_1 in the i^{th} position of the list of children of n_0 right-shifting by one all the children of n_0 whose indexes are greater than or equal to i. The primitive remove expects as input a node n and an integer i and removes the i^{th} child of n from its list of children left-shifting the right siblings of the removed node by one position. When given as input a node, the keyword len evaluates to its number of children. The keyword value expects as input a node and evaluates to the value that it stores, whereas the keyword store expects as input a node n and a value v and stores v in n. Finally, the syntax of Core DOM includes two special primitives: **(1)** end [12,14] (not available for the programmer) that is used by the semantics to signal the end of a structure block and **(2)** an internal while primitive used by the semantics for bookkeeping the guard of the loop currently executing.

We model a DOM forest $f : \mathcal{R}ef \rightarrow \mathcal{N}$ as a partial mapping from a set of references to the set of DOM nodes. A DOM node is a tuple of the form: $\langle m, \underline{v}, r, \omega \rangle$, where: **(1)** m is the node's tag, **(2)** $\underline{v}$ the runtime value that it stores, **(3)** r the reference pointing to its parent, and **(4)** ω its list of children. For simplicity, given a DOM node n, we denote by $n.\mathsf{tag}$, $n.\mathsf{value}$, $n.\mathsf{parent}$, and $n.\mathsf{children}$ its tag, value, parent, and list of children, respectively. The semantics of Core DOM makes use of a semantic function $\mathcal{R}_{Ancestor}$ that, given a forest f, outputs a binary relation in $\mathcal{R}ef \times \mathcal{R}ef$ such that $\langle r_0, r_1 \rangle \in \mathcal{R}_{Ancestor}(f)$ iff the node pointed to by r_0 is an ancestor of that pointed to by r_1. The set V_F contains the runtime values that cause the guard of a conditional or loop to fail.

Figures 1 and 2 present the small-step semantics of Core DOM. Configurations have the form: $\langle \mu, f, e \rangle$, where μ is a mapping from variables to values, f

$$\textsc{Move Upward}\quad \frac{f(r).\mathsf{parent} = r'}{\langle \mu, f, \mathsf{move}_{\uparrow}(r)\rangle \overset{\uparrow(r)}{\rightarrow} \langle \mu, f, r'\rangle} \qquad \textsc{Move Downward}\quad \frac{f(r).\mathsf{children}(i) = r'}{\langle \mu, f, \mathsf{move}_{\downarrow}(r,i)\rangle \overset{\downarrow(r')}{\rightarrow} \langle \mu, f, r'\rangle}$$

$$\textsc{New}\quad \frac{r = fresh(f, \sigma_0) \qquad f' = f\left[r \mapsto \langle m, null, null, \epsilon\rangle\right]}{\langle \mu, f, \mathsf{new}^{\sigma_0,\sigma_1,\sigma_2}(m)\rangle \overset{\mathsf{new}(r,\sigma_0,\sigma_1,\sigma_2)}{\rightarrow} \langle \mu, f', r\rangle}$$

$$\textsc{Remove}\quad \frac{f(r) = \langle m, \underline{v}, \hat{r}, \omega\rangle \qquad f(\omega(i)) = \langle m', \underline{v}', r, \omega'\rangle \qquad r' = \omega(i) \qquad f' = f\begin{bmatrix} r \mapsto \langle m, \underline{v}, \hat{r}, Shift_L(\omega, i)\rangle, \\ r' \mapsto \langle m', \underline{v}', null, \omega'\rangle \end{bmatrix}}{\langle \mu, f, \mathsf{remove}(r,i)\rangle \overset{-(r,r')}{\rightarrow} \langle \mu, f', r'\rangle}$$

$$\textsc{Insert}\quad \frac{\langle r', r\rangle \notin \mathcal{R}_{Ancestor}(f) \qquad f(r) = \langle m, \underline{v}, \hat{r}, \omega\rangle \qquad f(r') = \langle m', \underline{v}', null, \omega'\rangle \qquad f' = f\left[r \mapsto \langle m, \underline{v}, \hat{r}, Shift_R(\omega, i, r')\rangle, r' \mapsto \langle m', \underline{v}', r, \omega'\rangle\right]}{\langle \mu, f, \mathsf{insert}(r, r', i)\rangle \overset{+(r,r',\omega(i))}{\rightarrow} \langle \mu, f', r'\rangle}$$

$$\textsc{Length}\quad \frac{i = |f(r).\mathsf{children}|}{\langle \mu, f, \mathsf{len}(r)\rangle \overset{\mathsf{len}(r)}{\rightarrow} \langle \mu, f, i\rangle} \qquad \textsc{Value}\quad \frac{f(r).\mathsf{value} = \underline{v}}{\langle \mu, f, \mathsf{value}(r)\rangle \overset{\mathsf{val}(r)}{\rightarrow} \langle \mu, f, \underline{v}\rangle} \qquad \textsc{Store}\quad \frac{f(r) = \langle m, \underline{v}, \hat{r}, \omega\rangle \qquad f' = f\left[r \mapsto \langle m, \underline{v}', \hat{r}, \omega\rangle\right]}{\langle \mu, f, \mathsf{store}(r, \underline{v}')\rangle \overset{\mathsf{store}(r)}{\rightarrow} \langle \mu, f', \underline{v}'\rangle}$$

Fig. 2. Core DOM Semantics - Primitives for Tree Operations

a DOM forest, and e the expression to evaluate. References can be viewed as pointers to nodes, in the sense that the creation of a node yields a new reference that points to it. As in [6], the semantics makes use of a *parametric allocator*, $fresh$, that given a DOM forest f and a security level σ, generates a new reference r, such that $r \notin dom(f)$. The semantic transitions are annotated with *internal events* [12] to be used by the monitored semantics.

The evaluation order is specified by writing expressions using *evaluation contexts*. We write $E[e]$ to denote the expression obtained by replacing the occurrence of [] in the context E with e. The syntax of evaluation contexts is given by:

$$\begin{aligned} E ::= {} & [] \mid x = E \mid \mathsf{if}(E)\{e\}\ \mathsf{else}\ \{e\} \mid \mathsf{while}^{e}(E)\{e\} \mid E; e \mid \mathsf{end}(E) \mid \mathsf{len}(E) \mid \mathsf{value}(E) \\ & \mid \mathsf{move}_{\uparrow}(E) \mid \mathsf{move}_{\downarrow}(E, e) \mid \mathsf{move}_{\downarrow}(\underline{v}, E) \mid \mathsf{remove}(E, \underline{v}) \mid \mathsf{remove}(\underline{v}, E) \mid \mathsf{insert}(\underline{v}, E, e) \\ & \mid \mathsf{insert}(E, e, e) \mid \mathsf{insert}(\underline{v}, \underline{v}, E) \mid \mathsf{new}^{\sigma_0,\sigma_1,\sigma_2}(E) \mid \mathsf{store}(E, e) \mid \mathsf{store}(\underline{v}, E) \end{aligned}$$

Given a list ω, an integer i, and an arbitrary element a, we denote by: **(1)** $\omega(i)$ the i^{th} element of ω if it is defined and *null* otherwise, **(2)** $|\omega|$ the number of elements of ω, **(3)** ϵ the empty list, **(4)** $Shift_L(\omega, i)$ the list obtained by removing from ω its i^{th} element (provided that it is defined), **(5)** $Shift_R(\omega, a, i)$ the list obtained by inserting a in the i^{th} position of ω (provided that i is smaller than or equal to the number of elements of ω), **(6)** $\omega :: a$ the list obtained by appending a to ω, and **(7)** $\omega_0 \oplus \omega_1$ the concatenation of ω_0 and ω_1. We use the notation $f\left[a \mapsto b\right]$ for the function that coincides with f everywhere except in a that it maps to b.

3 Dynamic IFC in Core DOM

Before proceeding to describe the monitor for securing information flow in Core DOM, we discuss the main challenges imposed by the particular features of this

API and how we propose to tackle them. As usual, the specification of security policies relies on a lattice $\mathcal{L}$ of security levels and a labeling that maps resources to security levels. In examples and informal explanations, we use $\mathcal{L} = \{H, L\}$ with $L \sqsubseteq H$, meaning that resources labeled L (*low*, *visible*) are less confidential than those labeled H (*high*, *invisible*). Hence, H-labeled resources can depend on L-labeled resources, but not the contrary, as that would entail a *security leak*.

3.1 Challenges for IFC in Core DOM

The range of tree operations offered by Core DOM allows information to be stored and inspected from arbitrary nodes in several ways: **(1)** A node can be created and its existence tested; **(2)** A value can be stored and read from a node; **(3)** A child node can be inserted at/removed from a certain position, and both the number of children and their positions can be retrieved. (The *position* of a node can be understood as the pair consisting of its parent in the DOM forest and its index.) These operations can be used to encode security leaks via the different information components that are associated with every node. We now examine these leaks and introduce the formal techniques we use for tackling them. In the examples, we assume three initial nodes, div_0, div_1 and div_2, created as follows:

$$div_0 = \mathsf{new}(\text{"DIV"});\ div_1 = \mathsf{new}(\text{"DIV"});\ div_2 = \mathsf{new}(\text{"DIV"}) \tag{1}$$

Differentiating information components. Each node in a DOM forest can be seen to carry four main information components: its existence, its value, its position and its number of children. To some degree, these components can be manipulated separately, and there is value in treating them separately by the security analysis. For instance, in the following program, the final position of div_2 carries *high* information (because it is inserted in a *high* context), despite the fact that it contains the *low* level value l_0:

$$\mathsf{store}(div_2, l_0);\ \mathsf{if}(h)\{\mathsf{insert}(div_0, div_2, 0)\}\ \mathsf{else}\ \{\mathsf{insert}(div_1, div_2, 0)\} \tag{2}$$

After the execution of this program, the position of div_2 should not be revealed to a low observer. Its value, however, can be made public. Hence, while the evaluation of $\mathsf{move}_{\uparrow}(div_2)$ should yield a high value, the evaluation of the expression $\mathsf{value}(div_2)$ in the final memory can yield a low value. Similarly, there is no reason why the position of a node that stores a secret value should not be public.

By treating tree nodes as first-class values, we can naturally differentiate the security levels that are associated to each of the node's information components. We propose to associate every tree node with four security levels. The *value level* of a node is the level of the value that it stores. The *position level* of a node is the level of its position in the DOM forest. Hence, the position level of a node constitutes an upper bound on the levels of the contexts in which its position in the DOM forest can change (such as by its insertion/removal). The *structure security level* [10] of a node is associated to the node's number of children. It serves as an upper bound on the levels of the contexts in which the number of children of a node can be changed (such as by insertion/removal of nodes in/from its list of children). Finally, the *node level* is the level associated to information about the existence of the node itself. It is used as an upper bound on the levels

of the contexts in which the node can be created or a lower bound on its own value and structure level, and on its children's position levels.

New forms of security leaks. When inserting/removing a node in/from the list of children of a given node, the indexes of its right siblings change, thereby entailing a new kind of implicit flow. Consider the following example:

$$\mathsf{insert}(div_0, div_1, 0);\ \mathsf{if}(h)\{\mathsf{insert}(div_0, div_2, 0)\}\ \mathsf{else}\ \{null\};\ l_0 = \mathsf{move}_{\downarrow}(div_0, 0) \quad (3)$$

The program above prepends div_1 to the list of children of div_0 (which is originally empty). Then, depending on the value of h, the program prepends div_2 to the list of children of div_0. Hence, depending on the value of h, the program assigns either div_1 or div_2 to the *low* variable l_0. We refer to these forms of security leaks as *order leaks*, as they leverage information about the order of the nodes in the list of children. Order leaks can also be obtained by removing a child node when a second sibling node of higher index exists. Program 3 shows that, when changing the position of a node in the DOM forest, the positions of its right siblings also change. Therefore, the monitor enforces the position levels of the right siblings of a given node to be equal to or higher than its own position level. Furthermore, when moving from one node to another information about the position of the child node is leaked. For instance, for Program 3 to be legal, the position levels of div_1 and div_2 must be H. Therefore, the value associated with the evaluation of the instruction $\mathsf{move}_{\downarrow}(div_0, 0)$ is H.

The fact that a program can inspect the number of children of a given node can be exploited to encode implicit information flows. If we add the low assignment $l_1 = \mathsf{len}(div_0)$ to the end of Program 3, the value of l_1 will be set to 2 or 1 depending on the value of the *high* variable h. The *structure security level* is meant to control this kind of leaks. If a node has *low* structure security level, one cannot insert/remove nodes in/from its list of children in *high* contexts. Therefore, the level associated with looking-up the number of children of a given node corresponds to its structure security level. For instance, for Program 3 to be legal, the structure security level of div_0 must be H. Hence, the level associated with the evaluation of $\mathsf{len}(div_0)$ is H independently of the original value of h.

Flow-sensitive versus flow-insensitive IF monitoring. The *no-sensitive-upgrade* discipline [3] has been widely used in the design of *purely dynamic monitors*. This discipline establishes that visible resources cannot be upgraded in invisible contexts, since such upgrades cause the visible domain of a program to change depending on secret values. Hence, flow-sensitive monitors that implement the no-sensitive-upgrade discipline abort executions that encode illegal implicit flows. Since both the structure security level and the position level of a node are used to control the implicit flows that can be encoded by inserting/removing nodes in/from the DOM forest, these levels cannot be upgraded. This point is illustrated in the following table, which represents four monitored executions of a program (represented on the left) in two distinct memories, by showing how the variable labeling Γ and the node labeling Σ evolve during each execution. The initial memories are such that div_0 and div_1 each bind an orphan node with *low* structure security level, and are pointed to by r_0 and r_1, respectively, but differ in the value of *high* variable h.

While the monitor following the *naive approach* raises the structure security level of div_0 to H (allowing the execution to go through), the monitor following the *no-sensitive-upgrade* discipline blocks the execution when the program tries to add div_1 to the list of children of div_0 in a *high* context. The case regarding the position level can be seen by replacing the test of the second `if` instruction with $\mathsf{move}_{\uparrow}(div_1) == div_0$, assuming that the original position level of div_1 is *low*. In contrast to the position level and to the structure security level, the value level of a node can be upgraded, as the value stored in a node is set explicitly. However, such upgrades cannot be caused by implicit information flows.

	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
Initial High Memory:	$h = \mathsf{false}$	$h = \mathsf{true}$	$h = \mathsf{true}$
$l = \mathsf{true}$	$\Gamma(l) := L$	$\Gamma(l) := L$	$\Gamma(l) := L$
$\mathsf{if}(h)$	branch not taken	branch taken	branch taken
$\mathsf{insert}(div_0, div_1, 0)$	—	$\Sigma(r_0).\mathsf{struct} := H$	*stuck*
$\mathsf{if}(\mathsf{len}(div_0) == 0)$	branch taken	branch not taken	—
$l = \mathsf{false}$	$\Gamma(l) := L$	—	—
Final Low Memory:	$l = \mathsf{false}$	$l = \mathsf{true}$	—

3.2 The Attacker Model

We assume a generic lattice $\mathcal{L}$ of security levels, and use $\sqcup$, $\sqcap$, $\bot$, and $\top$ for the greatest lower bound, the least upper bound, the *bottom* level, and the *top* level, respectively. We consider two types of labelings: *variable labelings* and *node labelings*. While a variable labeling $\Gamma : \mathcal{V}ar \to \mathcal{L}$ maps each variable to a single security level, a node labeling $\Sigma : \mathcal{R}ef \to \mathcal{L}^4$ associates each reference with a tuple of four security levels. Hence, given a reference r and a labeling Σ, $\Sigma(r) = \langle \sigma_n, \sigma_v, \sigma_e, \sigma_s \rangle$, where: **(1)** σ_n is the node level, **(2)** σ_v is the value level, **(3)** σ_e is the the position level, and **(4)** σ_s is the structure security level. For clarity, given a node n pointed to by a reference r and a labeling Σ, we denote by $\Sigma(r).\mathsf{node}$, $\Sigma(r).\mathsf{value}$, $\Sigma(r).\mathsf{pos}$, and $\Sigma(r).\mathsf{struct}$ its node level, value level, position level, and structure security level, respectively. For simplicity, we impose four restrictions on the levels assigned to a given node. First, one cannot store a visible value in an invisible node. Second, an invisible node cannot have a visible position. Third, an invisible node cannot have a visible number of children. Fourth, an invisible node cannot have a visible node in its list of children (in practice, this means that we cannot insert a visible node in an invisible node). Formally, for every reference $r \in dom(\Sigma)$, it holds that: $\Sigma(r).\mathsf{node} \sqsubseteq \Sigma(r).\mathsf{value} \sqcap \Sigma(r).\mathsf{pos} \sqcap \Sigma(r).\mathsf{struct}$. Additionally, for every two references r and r' in a forest f such that: $r, r' \in dom(\Sigma)$ and $f(r).\mathsf{children}(i) = r'$ for some integer i, it holds that $\Sigma(r).\mathsf{node} \sqsubseteq \Sigma(r').\mathsf{node}$.

In order to formally characterize the observational power of an attacker, we take the standard approach of defining a notion of *low-projection* of a memory/forest at a given level σ, which corresponds to the part of the memory/-forest that an attacker at level σ can observe. The low-projection of a memory μ with respect to a variable labeling Γ at level σ is simply given by: $\mu \restriction^{\Gamma,\sigma} = \{(x, \mu(x), \Gamma(x)) \mid x \in dom(\Gamma) \wedge \Gamma(x) \sqsubseteq \sigma\}$. Accordingly, two memories μ_0 and μ_1, respectively labeled by Γ_0 and Γ_1 are said to be *low-equal* at

$$\textsc{Identifier}\quad \frac{\sigma = level(o) \sqcup \Gamma(x)}{\langle \Gamma, \Sigma, o, \zeta\rangle \overset{\mathsf{var}(x)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle} \qquad \textsc{Assignment}\quad \frac{level(o) \sqsubseteq \Gamma(x) \qquad \sigma' = level(o) \sqcup \sigma}{\langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\mathsf{assign}(x)}{\rightarrow} \langle \Gamma\left[x \mapsto \sigma'\right], \Sigma, o, \zeta :: \sigma'\rangle}$$

$$\textsc{Literal Value}\quad \langle \Gamma, \Sigma, o, \zeta\rangle \overset{\mathsf{pval}}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: level(o)\rangle \qquad \textsc{Branch}\quad \langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\mathsf{branch}}{\rightarrow} \langle \Gamma, \Sigma, o :: \sigma, \zeta\rangle$$

$$\textsc{End}\quad \langle \Gamma, \Sigma, o :: \sigma, \zeta\rangle \overset{\succ}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta\rangle \qquad \textsc{Discharge}\quad \langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\bullet}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta\rangle \qquad \textsc{Empty}\quad \langle \Gamma, \Sigma, o, \zeta\rangle \overset{\circ}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta\rangle$$

Fig. 3. Core DOM Monitor - Imperative Fragment

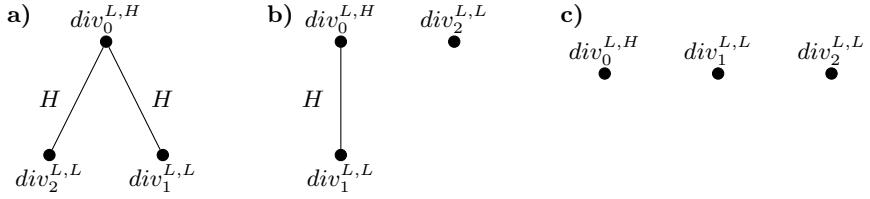
security level σ, written $\mu_0, \Gamma_0 \sim_\sigma \mu_1, \Gamma_1$, if they coincide in their respective low-projections, $\mu_0 \restriction^{\Gamma_0,\sigma} = \mu_1 \restriction^{\Gamma_1,\sigma}$. Definition 1 extends the notion of low-projection to forests. Informally, an attacker at level σ can see: **(1)** the references whose corresponding nodes are associated with levels $\sqsubseteq \sigma$ as well as their tags, **(2)** the values stored in visible nodes whose value level is $\sqsubseteq \sigma$, **(3)** the positions of visible nodes (with visible parents) whose levels are $\sqsubseteq \sigma$, and **(4)** the number of children of visible nodes whose structure security level is $\sqsubseteq \sigma$.

Definition 1 (Low-Projection and Low-Equality). *The low-projection of a forest f w.r.t. a security level σ and a labeling Σ is given by:*

$$\begin{aligned} f \restriction^{\Sigma,\sigma} = \ & \{(r, f(r).\mathsf{tag}, \Sigma(r).\mathsf{node}, \Sigma(r).\mathsf{pos}, \Sigma(r).\mathsf{struct}) \mid \Sigma(r).\mathsf{node} \sqsubseteq \sigma\} \\ \cup \ & \{(r, f(r).\mathsf{value}, \Sigma(r).\mathsf{value}) \mid \ \Sigma(r).\mathsf{value} \sqsubseteq \sigma\} \\ \cup \ & \{(r, i, r') \mid \ f(r).\mathsf{children}(i) = r' \ \wedge \ \Sigma(r').\mathsf{pos} \sqsubseteq \sigma\} \\ \cup \ & \{(r, null) \mid \ f(r).\mathsf{parent} = null \ \wedge \ \Sigma(r).\mathsf{pos} \sqsubseteq \sigma\} \\ \cup \ & \{(r, |f(r).\mathsf{children}|) \mid \ \Sigma(r).\mathsf{struct} \sqsubseteq \sigma\} \end{aligned}$$

Two forests f_0 and f_1, respectively labeled by Σ_0 and Σ_1 are said to be low-equal at security level σ, written $f_0, \Sigma_0 \sim_\sigma f_1, \Sigma_1$, if $f_0 \restriction^{\Sigma_0,\sigma} = f_1 \restriction^{\Sigma_1,\sigma}$.

In the following figures, **a)** and **b)** represent the final forests obtained from the execution of Program 3 in two distinct memories that initially map the *high* variable h to 1 and to 0, respectively. Forest **c)** represents their (coinciding) low-projection. Nodes are labeled with their node level and structure security level, while edges are labeled with the child's position level. The position levels of div_1 and div_2 as well as the structure security level of div_0 are assumed to be originally *high*. All other levels are assumed to be originally *low*.



3.3 Enforcement

We define a monitored semantics in the style of Russo et al. [12, 14]. A configuration of the monitored semantics is obtained by pairing up a configuration

of the unmonitored semantics with a configuration of the security monitor. At each computation step, the security monitor uses the internal event generated by the unmonitored semantics to determine its corresponding transition. Hence, the transitions of the monitored semantics are defined as follows:

$$\frac{\langle \mu, f, e\rangle \xrightarrow{\alpha} \langle \mu', f', e'\rangle \quad \langle \Gamma, \Sigma, o, \zeta\rangle \xrightarrow{\alpha} \langle \Gamma', \Sigma', o', \zeta'\rangle}{\langle\langle \mu, f, e\rangle, \langle \Gamma, \Sigma, o, \zeta\rangle\rangle \rightarrow \langle\langle \mu', f', e'\rangle, \langle \Gamma', \Sigma', o', \zeta'\rangle\rangle}$$

The arrow $\rightarrow$ denotes the transition relation (whilst $\rightarrow^*$ its reflexive-transitive closure), and the configurations of the security monitor have the form $\langle \Gamma, \Sigma, o, \zeta\rangle$, where: **(1)** Γ is the variable labeling, **(2)** Σ is the node labeling, **(3)** o is the *control context*, that is, a list containing the levels of the expressions on which the program branched in order to reach the expression that is currently executing, and **(4)** ζ is the *expression context*, that is, a list consisting of the levels of the expressions of the current evaluation context that were already computed. The control context and the expression context lists are used as stacks. Concretely, each time the evaluation enters the body of an `if` or a `while` expression, the level of the expression that was tested is appended to the control context. Conversely, when the control flow leaves the body of an `if` or a `while` expression, the monitor removes the last element of the control context. Similarly, after the evaluation of an expression that is nested inside another expression, its level is appended to the expression context and, whenever an expression is no longer nested inside another expression, its level is removed from the expression context. The transitions of the monitor are presented in Figures 3 and 4. We are only concerned with monitored executions beginning with $\langle \Gamma_0, \Sigma_0, \epsilon, \epsilon\rangle$, where Γ_0 and Σ_0 are the original variable and node labelings. Furthermore, letting μ_0 and f_0 be the initial memory and the initial forest, we require that: $dom(\mu_0) = dom(\Gamma_0)$ and $dom(f_0) = dom(\Sigma_0)$. Given a list ω, we use the $level(\omega)$ as an abbreviation for $\sqcup\{\omega(i) \mid 0 \leq i < |\omega|\}$.

Let us briefly explain the rules of the proposed information flow monitor. Rules [Move Upward] and [Move Downward] update the expression context ζ with the levels of the current expression's subexpression(s) (that are retrieved from ζ), of the program counter, and of the departing/arriving node's position. Observe that the levels of the nodes that the traversed edge connects are ignored, as they are assumed to be lower than or equal to the child's position level. Analogously, rule [Length] updates the expression context ζ with the levels of the current expression's subexpression, of the program counter, and of the structure security level of the node whose number of children is being inspected. Rules [Remove] and [Insert] prevent the removal/insertion of a node with a visible position in an invisible context. Furthermore, they prevent the semantics from inserting/removing a node in/from a node with a visible number of children in an invisible context. Since changing the position of a node causes the position of its right siblings to change, Rule [Insert] ensures that the position levels of the children of every DOM node are always in increasing order. Finally, rules [Store] and [Assignment] update the security level of the corresponding value, provided that it does not constitute a sensitive-upgrade. Hence, updates of visible values in invisible contexts cause the execution to abort.

$$\textsc{Move Upward}\quad \frac{\sigma' = level(o) \sqcup \Sigma(r).\mathsf{pos} \sqcup \sigma}{\langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\uparrow(r)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma'\rangle} \qquad \textsc{Move Downward}\quad \frac{\sigma' = level(o) \sqcup \Sigma(r).\mathsf{pos} \sqcup \sigma_0 \sqcup \sigma_1}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1\rangle \overset{\downarrow(r)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma'\rangle}$$

$$\textsc{Length}\quad \frac{\sigma' = \sigma \sqcup level(o) \sqcup \Sigma(r).\mathsf{struct}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\mathsf{len}(r)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma'\rangle} \qquad \textsc{Remove}\quad \frac{level(o) \sqcup \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma(r).\mathsf{struct} \sqcap \Sigma(r').\mathsf{pos}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1\rangle \overset{-(r,r')}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \Sigma(r').\mathsf{pos}\rangle}$$

$$\textsc{Insert}\quad \frac{\begin{array}{c} r'' = null \vee \Sigma(r').\mathsf{pos} \sqsubseteq \Sigma(r'').\mathsf{pos} \\ level(o) \sqcup \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Sigma(r).\mathsf{struct} \sqcap \Sigma(r').\mathsf{pos}\end{array}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1 :: \sigma_2\rangle \overset{+(r,r',r'')}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \Sigma(r').\mathsf{pos}\rangle} \qquad \textsc{Value}\quad \frac{\sigma' = \sigma \sqcup \Sigma(r).\mathsf{value}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\mathsf{val}(r)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma'\rangle}$$

$$\textsc{Store}\quad \frac{\begin{array}{c} \sigma = level(o) \sqcup \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r).\mathsf{node} \\ level(o) \sqcup \sigma_0 \sqsubseteq \Sigma(r).\mathsf{value} \\ \Sigma' = \Sigma\,[r \mapsto \langle \Sigma(r).\mathsf{node}, \sigma, \Sigma(r).\mathsf{pos}, \Sigma(r).\mathsf{struct}\rangle]\end{array}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1\rangle \overset{\mathsf{store}(r)}{\rightarrow} \langle \Gamma, \Sigma', o, \zeta :: \sigma_1\rangle} \qquad \textsc{New}\quad \frac{\begin{array}{c} level(o) \sqcup \sigma \sqsubseteq \sigma_0 \sqsubseteq \sigma_1 \sqcap \sigma_2 \\ \Sigma' = \Sigma\,[r \mapsto \langle \sigma_0, \sigma_0, \sigma_1, \sigma_2\rangle]\end{array}}{\langle \Gamma, \Sigma, o, \zeta :: \sigma\rangle \overset{\mathsf{new}(r,\sigma_0,\sigma_1,\sigma_2)}{\rightarrow} \langle \Gamma, \Sigma', o, \zeta :: \sigma_0\rangle}$$

Fig. 4. Core DOM Monitor - Primitives for Tree Operations

Informally, a monitor is said to be *noninterferent* (NI) if, whenever the monitored execution of a program on two low-equal memories/forests terminates successfully, it also produces two low-equal memories/forests. Hence, an attacker cannot use the monitored execution of a program as a means to disclose information about the confidential contents of a memory. Theorem 1 states that the monitored successfully-terminating execution of a program on two low-equal memories/forests always yields two low-equal memories/forests.

Theorem 1 (Noninterference). *For any expression e, memories μ_0 and μ_1, forests f_0 and f_1, variable labelings Γ_0 and Γ_1, node labelings Σ_0 and Σ_1, and security level σ such that $\mu_0, \Gamma_0 \sim_\sigma \mu_1, \Gamma_1$ and $f_0, \Sigma_0 \sim_\sigma f_1, \Sigma_1$, and also:*

- $\langle\langle \mu_0, f_0, e\rangle, \langle \Gamma_0, \Sigma_0, \epsilon, \epsilon\rangle\rangle \rightarrow^* \langle\langle \mu_0', f_0', v_0\rangle, \langle \Gamma_0', \Sigma_0', \epsilon, \epsilon :: \sigma_0\rangle\rangle$
- $\langle\langle \mu_1, f_1, e\rangle, \langle \Gamma_1, \Sigma_1, \epsilon, \epsilon\rangle\rangle \rightarrow^* \langle\langle \mu_1', f_1', v_1\rangle, \langle \Gamma_1', \Sigma_1', \epsilon, \epsilon :: \sigma_1\rangle\rangle$

It holds that: $\mu_0', \Gamma_0' \sim_\sigma \mu_1', \Gamma_1'$ and $f_0', \Sigma_0' \sim_\sigma f_1', \Sigma_1'$. Furthermore, if either $\sigma_0 \sqsubseteq \sigma$ or $\sigma_1 \sqsubseteq \sigma$, then: $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ and $v_0 = v_1$.

4 Extension to Live Primitives

The DOM API includes several methods that return live collections. For instance, the method `getElementsByTagName` returns a live collection containing all the nodes in the document tree whose tag matches the string given as input. The distinctive feature of live collections is that they automatically reflect modifications to the document. Hence, every time a node matching the query that generated a given live collection is inserted/removed in/from the document, it is also automatically inserted/removed in/from the corresponding live collection. Therefore, a live collection is in fact a dynamic query to the document.

$$\textsc{Live Move}\quad \frac{f \vdash r \rightsquigarrow_m \omega \qquad \omega(i) = r'}{\langle \mu, f, \mathsf{move}_\ell\,(r, m, i)\rangle \overset{\ell\,(f,r')}{\rightarrow} \langle \mu, f, r'\rangle} \qquad \textsc{Live Length}\quad \frac{f \vdash r \rightsquigarrow_m \omega}{\langle \mu, f, \mathsf{len}_\ell\,(r, m)\rangle \overset{\mathsf{len}_\ell\,(f,r,m)}{\rightarrow} \langle \mu, f, |w|\rangle}$$

Fig. 5. Extension of the Semantics to Live Primitives

4.1 Formal Syntax and Semantics

The nodes of a live collection are always arranged in *document order*. The document order is an ordering $\leq$ on the nodes of the DOM forest such that for every two nodes n_0 and n_1 in the same DOM tree, $n_0 \leq n_1$ if and only if n_0 is found before n_1 in a depth-first left-to-right search starting from the root of the tree. In order to model the live collections returned by the method `getElementsByTagName`, we extend Core DOM with two additional constructs:

$$e ::= ... \mid \mathsf{move}_\ell\,(e, e, e) \mid \mathsf{len}_\ell\,(e, e)$$

The *live move* primitive receives as input a node n, a tag name m, and an integer i and evaluates to the i^{th} node with tag m in the tree rooted at n when traversed in document order. The *live length* primitive receives as input a node n and a tag name m and evaluates to the number of nodes with tag m in the tree rooted at n. The example given in Section 1 can be rewritten in Core DOM as:

$$i = 0;\ \mathsf{while}(i < \mathsf{len}_\ell\,(\mathsf{doc}, \text{“}DIV\text{”}))\{\mathsf{insert}(\mathsf{doc}, \mathsf{new}(\text{“}DIV\text{”}), \mathsf{len}(\mathsf{doc}));\ i = i + 1\} \quad (4)$$

where doc is assumed to be a special identifier bound to the root of the *document*. The syntax of the evaluation contexts is extended to take into account the new syntactic constructs:

$$E ::= ... \mid \mathsf{move}_\ell\,(E, e, e) \mid \mathsf{move}_\ell\,(\underline{v}, E, e) \mid \mathsf{move}_\ell\,(\underline{v}, \underline{v}, E) \mid \mathsf{len}_\ell\,(E, e) \mid \mathsf{len}_\ell\,(\underline{v}, E)$$

The extension of the semantics to live primitives is presented in Figure 5. The semantics makes use of a search predicate of the form $f \vdash r \rightsquigarrow_m \omega$ (given in appendix), that formalizes the search for the nodes matching a given tag in a tree. Intuitively, given a forest f, a reference to a node r, a tag name m, and a list of DOM references ω, $f \vdash r \rightsquigarrow_m \omega$ holds iff ω is the list of all the nodes with tag m found when traversing the tree of f rooted at r in document order.

4.2 Information Leaks due to Live Primitives

The *live* constructs introduced in this section can be exploited to encode new types of information leaks.

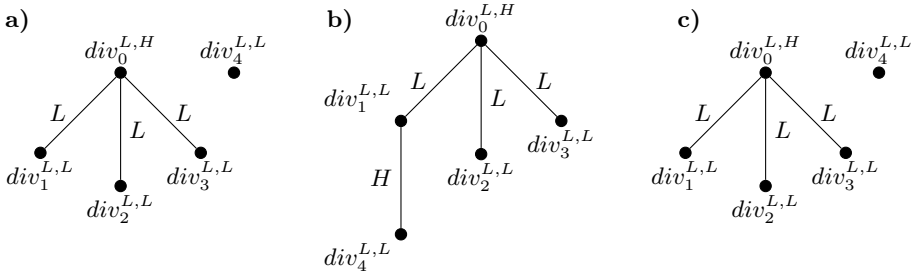
Leaks via len_ℓ. Consider the program below, which is to be executed in a forest that originally contains five orphan **DIV** nodes respectively bound to the variables: div_0, div_1, div_2, div_3, and div_4.

$$\begin{array}{l}\mathsf{insert}(div_0, div_1, 0);\ \mathsf{insert}(div_0, div_2, 1);\ \mathsf{insert}(div_0, div_3, 2);\\ \mathsf{if}(h)\{\mathsf{insert}(div_1, div_4, 0)\}\ \mathsf{else}\ \{null\};\ l = \mathsf{len}_\ell\,(div_0)\end{array} \quad (5)$$

Depending on the value of h, l may be set either to 4 or 5. In order to tackle this type of leak, we require the programmer to pre-establish for each possible tag name m an upper bound on the position levels of the nodes with that tag name, that we denote by σ_m and call *global position level*. For instance, $\sigma_{\textbf{DIV}}$

corresponds to the pre-established upper bound on the position levels of DIV nodes. In order to allow the execution of $\mathsf{len}_{\not l}$ to go through, the monitor checks whether the position levels of all nodes in the forest are lower than or equal to the global position level, in which case the level associated with the expression is the global position level. Therefore, for Program 5 to be legal $\sigma_{\mathbf{DIV}}$ must be set to H. Accordingly, the expression $\mathsf{len}_{\not l}(div_0)$ yields a value of level H.

Leaks via $\mathsf{move}_{\not l}$. As the primitive $\mathsf{move}_{\not l}$ traverses the tree in document order, it can be used to encode a new type of *order leak*. Let us modify Program 5 by replacing the last instruction with $l = \mathsf{move}_{\not l}(div_0, \text{“}DIV\text{”}, 3)$. Then, depending on the value of the *high* variable h, l is assigned to div_3 or div_2. A NI monitor must detect this information flow and raise the level of the originally *low* variable l to H. In particular, for this program to be legal (according to the current enforcement mechanism), the position level of div_4 as well as the structure security level of div_1 must be *high*. All other levels can be set to L. In the following figures, **a)** and **b)** represent the final forests obtained from the execution of this program in two distinct memories that initially map the h to 0 and to 1, respectively. Forest **c)** represents their (coinciding) low-projection. Observe that in spite of being evaluated in two low-equal memories and only manipulating visible values, the evaluation of $\mathsf{move}_{\not l}(div_0, \text{“}DIV\text{”}, 3)$ yields two different values.



The key insight for securing the new information flows introduced by the $\mathsf{move}_{\not l}$ primitive is that this primitive allows an attacker to operate on the nodes with the same tag in the same tree as if they were siblings. Hence, it is necessary to adjust the notion of a node's position in order to take into account this new way of traversing the DOM forest. Let the *live index* of a node be its position in the list of nodes obtained by searching its corresponding tree for the nodes with its tag in document order. The position of a node is now understood as the triple consisting of its parent, its index, and its live index. Hence, changing the position of a node in a tree causes the positions of the nodes with the same tag in the same tree with higher live indexes to change. In order to deal with this kind of flow, the proposed enforcement mechanism guarantees that the execution of a live move only goes through if, for every tag name m and node n, the position levels of the nodes with tag m in the tree rooted at n monotonically increase in document order. For instance, in the figure above, the final forest **b)** obtained when $h = 1$ does not comply with this requirement because the position level of div_4 is not lower than or equal to the position level of div_2, while the live index of div_4 is lower than the live index of div_2.

4.3 Revised Attacker Model and Enforcement Mechanism

At the formal level, the introduction of the new live primitives poses two separate problems. First, the low-equality definition must be restated so as to correctly capture the observational power of an attacker disposing of these new primitives. Second, the monitor must be extended in such a way that it remains noninterferent. We modify the definition of low-projection so that an attacker at level σ can additionally see: **(1)** the live indexes of the nodes whose position levels are $\sqsubseteq \sigma$ and **(2)** the number of descendants of visible nodes with a given tag m such that $\sigma_m \sqsubseteq \sigma$. Formally:

$$f \restriction_\ell^{\Sigma,\sigma} = f \restriction^{\Sigma,\sigma} \cup \{(r,m,i,r') \mid\ f \vdash r \rightsquigarrow_m \omega\ \wedge\ \omega(i) = r'\ \wedge\ \Sigma(r').\mathsf{pos} \sqsubseteq \sigma\}$$
$$\cup\ \{(r,m,n) \mid\ f \vdash r \rightsquigarrow_m \omega\ \wedge\ |\omega| = n\ \wedge\ \sigma_m \sqcup \Sigma(r).\mathsf{node} \sqsubseteq \sigma\}$$

As expected, two labeled forests are low-equal at a given level σ, written $f_0, \Sigma_0 \sim_\sigma^\ell f_1, \Sigma_1$, if they coincide in their respective low-projections.

We do not modify the previous monitor so that the new low-equality is preserved by monitored executions. Instead, we establish a predicate – $\mathcal{WL}(f,\Sigma)$ – for labeled forests, such that any two labeled forests verifying this predicate and related by the first low-equality are also related by the new low-equality. This is formally stated as follows:

Theorem 2 (Low-Equality Strengthening). *Given two forests f_0 and f_1 respectively labeled by Σ_0 and Σ_1 and a security level σ such that $\mathcal{WL}(f_0,\Sigma_0)$ and $\mathcal{WL}(f_1,\Sigma_1)$ and $f_0,\Sigma_0 \sim_\sigma f_1,\Sigma_1$, it holds that: $f_0,\Sigma_0 \sim_\sigma^\ell f_1,\Sigma_1$.*

Informally, $\mathcal{WL}(f,\Sigma)$ holds if and only if: **(1)** the position levels of all the nodes in f are lower than or equal to the respective global position levels, **(2)** the position levels of the nodes with the same tag monotonically increase in document order, and **(3)** the position level of every node is higher than or equal to the position levels of all its descendants (meaning that if the position of a node is secret, the positions of all its descendants are also secret). The predicate $\mathcal{WL}(f,\Sigma)$ is defined with the help of a predicate $\mathcal{WL}_{f,\Sigma} \vdash^r \phi_\ell \rightsquigarrow \phi'_\ell$, defined below, that holds if the tree rooted at r is well-labeled. The function ϕ_ℓ maps each tag name to the position level of the last node with that tag name preceding the node pointed to by r in f in document order. The function ϕ'_ℓ maps each tag name to the position level of the last node with that tag name in the tree rooted at r (if no such node exists, ϕ'_ℓ coincides with ϕ_ℓ). Formally, the predicate $\mathcal{WL}(f,\Sigma)$ holds if and only if for all orphan nodes pointed to by a reference r there are two functions ϕ_ℓ and ϕ'_ℓ such that $\mathcal{WL}_{f,\Sigma} \vdash^r \phi_\ell \rightsquigarrow \phi'_\ell$.

$$\textsc{Orphan Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \\ |f(r).\mathsf{children}| = 0 \\ \phi_\ell(m) \sqsubseteq \Sigma(r).\mathsf{pos} \sqsubseteq \sigma_m \\ \phi'_\ell = \phi_\ell\,[m \mapsto \Sigma(r).\mathsf{pos}] \end{array}}{\mathcal{WL}_{f,\Sigma} \vdash^r \phi_\ell \rightsquigarrow \phi'_\ell}$$

$$\textsc{Non-Orphan Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \qquad \phi_\ell(m) \sqsubseteq \Sigma(r).\mathsf{pos} \sqsubseteq \sigma_m \\ |f(r).\mathsf{children}| = n > 0 \qquad \phi^0_\ell = \phi_\ell\,[m \mapsto \Sigma(r).\mathsf{pos}] \\ \forall_{0\le i<n}\ \Sigma(r).\mathsf{pos} \sqsubseteq \Sigma(f(r).\mathsf{children}(i)).\mathsf{pos} \\ \forall_{0\le i<n}\ \mathcal{WL}_{f,\Sigma} \vdash^{f(r).\mathsf{children}(i)} \phi^i_\ell \rightsquigarrow \phi^{i+1}_\ell \end{array}}{\mathcal{WL}_{f,\Sigma} \vdash^r \phi_\ell \rightsquigarrow \phi^n_\ell}$$

Finally, the extension of the security monitor to the new live primitives is given in Figure 6. The evaluation of these primitives only goes through if the corresponding forest is well-labeled.

$$\textsc{Live Move}\quad \frac{\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Sigma(r).\mathsf{pos} \qquad \mathcal{WL}(f,\Sigma)}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1 :: \sigma_2 \rangle \overset{\natural(f,r)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma' \rangle}$$

$$\textsc{Live Length}\quad \frac{\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_m \sqcup \Sigma(r).\mathsf{node} \qquad \mathcal{WL}(f,\Sigma)}{\langle \Gamma, \Sigma, o, \zeta :: \sigma_0 :: \sigma_1 \rangle \overset{\mathsf{len}_\natural(f,r,m)}{\rightarrow} \langle \Gamma, \Sigma, o, \zeta :: \sigma' \rangle}$$

Fig. 6. Extension of the Monitor to Live Primitives

5 Related Work and Conclusions

The increasing popularity of scripting languages has motivated further research on runtime mechanisms for securing information flow, such as *monitors*. In contrast to *purely dynamic monitors* [3–5] that do not rely on any kind of static analysis, *hybrid monitors* [8, 15, 16] use static analysis to reason about the implicit flows that arise due to untaken execution paths. Given the dynamic nature of tree operations, designing such a static analysis for Core DOM is far from trivial. Hence, we have chosen to present a purely dynamic monitor and we leave the design of its hybrid version for future work.

Russo et al. [13] were the first to study the problem of securing information flow in DOM-like dynamic tree structures. They present a monitor for a WHILE language with primitives for manipulating DOM-like trees and prove it sound. However, references are not modeled in this language; instead, program configurations include the current working node of the program. This is, as the authors point out, the main difference with respect to JavaScript DOM operations, since in JavaScript tree nodes are treated as first-class values. In particular, in [13] it is not possible to change the position of a node in the DOM forest without deleting and re-creating it – its position remains the same during its whole "lifetime". Consequently, the *position level* of a node coincides with its *node level*. By treating nodes as first-class values we were able to give separate treatment to position leaks, which cannot be directly expressed in the language of [13].

Hedin et al. [9] implemented the first information flow monitor for fully-fledged JavaScript together with "statefull information-flow models" for the standard API, as well as several APIs that are present in a browser environment such as the DOM API. The presentation includes an informal explanation on how the problem of live collections returned by the method `getElementsByName` is dealt with. Their approach for dealing with live leaks coincides with the technique we employ to the particular case of the $\mathsf{len}_\natural$ primitive.

Gardner et al. [7] propose a compositional and concise formal specification of the DOM called Minimal DOM. The authors show that their semantics has no redundancy and that it is sufficient to describe the structural kernel of DOM Core Level 1. Additionally, they apply local reasoning based on Separation Logic and prove invariant properties of simple JavaScript programs that interact with the DOM. Given that our aim is to track information flow in the DOM, we use a simplified semantics that allows us to label DOM resources in a natural way. Like Minimal DOM, Core DOM is also compositional. Furthermore, all the primitives of Minimal DOM can be easily translated to Core DOM. Hence, we expect the authors' sufficiency claim to be applicable to Core DOM.

This paper contributes to the challenge of enforcing secure information flow in client-side Web applications by presenting a provably sound flow-sensitive security monitor that enforces noninterference over Core DOM, an expressive

representative subset of the DOM API. The proposed solution tackles open issues in IF security such as references and live collections in dynamic tree structures. By including references and live collections, Core DOM offers the expressive power of the DOM in the form of a simple language that is well tailored for automatic program analysis. We thus believe that it could be re-used in future research on security aspects of the DOM API.

Acknowledgments. This work was partially supported by the Portuguese Government via the PhD grant SFRH/BD/71471/2010.

References

1. The 5th edition of ECMA 262 June 2011. ECMAScript Language Specification. Technical report, ECMA, 2011.
2. A. Almeida Matos, J. Fragoso Santos, and T. Rezk. An IF monitor for a core of DOM. http://web.ist.utl.pt/∼ana.matos/14-AFR-if+monitor+coredom-full.pdf.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.
4. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.
5. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.
6. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *CSFW*, 2002.
7. P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. DOM: Towards a formal specification. In *PLAN-X*, 2008.
8. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
9. D. Hedin, B. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.
10. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF*, 2012.
11. W3C Recommendation. DOM: Document Object Model (DOM). Technical report, W3C, 2005.
12. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.
13. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
14. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*, 2009.
15. P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, 2007.
16. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS*, 2006.