Information Flow Control for Event Handling and the DOM in Web Browsers

Vineet Rajani
MPI-SWS, Germany
vrajani@mpi-sws.org

Abhishek Bichhawat
Saarland University, Germany
bichhawat@cs.uni-saarland.de

Deepak Garg
MPI-SWS, Germany
dg@mpi-sws.org

Christian Hammer
Saarland University, Germany
hammer@cs.uni-saarland.de

***Abstract*—Web browsers routinely handle private information. Owing to a lax security model, browsers and JavaScript in particular, are easy targets for leaking sensitive data. Prior work has extensively studied information flow control (IFC) as a mechanism for securing browsers. However, two central aspects of web browsers — the Document Object Model (DOM) and the event handling mechanism — have so far evaded thorough scrutiny in the context of IFC. This paper advances the state-of-the-art in this regard. Based on standard specifications and the code of an actual browser engine, we build formal models of both the DOM (up to Level 3) and the event handling loop of a typical browser, enhance the models with fine-grained taints and checks for IFC, prove our enhancements sound and test our ideas through an instrumentation of WebKit, an in-production browser engine. In doing so, we observe several channels for information leak that arise due to subtleties of the event loop and its interaction with the DOM.**

I. Introduction

A lot of confidential information like passwords, authentication cookies, credit card numbers, search queries and browsing history passes through web browsers. Client-side applications can easily read and leak or misuse this information, due to either malicious intent or insecure programming practices [1], [2], [3], [4]. Browser vendors are sensitive to this problem, but conventional data protection solutions implemented in web browsers have loopholes that can be, and often are, exploited. For example, the standard same-origin policy (SOP) [5], which is intended to restrict cross-domain data flows, can be easily bypassed by malicious programs through cross-domain image download requests that are exempt from the policy by design. This has often been exploited to leak cookies from webpages.

A significant source of security problems in web applications is the lax security model of the ubiquitous client-side programming language JavaScript (JS). In all browsers, third-party JS included in a page runs with the privileges of the page. This enables data leaks when untrustworthy third-party scripts are blindly included by website developers. Content-security policies (CSPs) [6] that allow whitelisting of trusted websites have been implemented to alleviate the problem, but CSPs also disable inline JS and dynamic code insertion (through the JS construct `eval`()), both of which are widely used [7]. More fine-grained data protection methods such as Google's Caja [8], FBJS [9] or AdSafe [10] use static analysis or source rewriting to limit access of third-party code to confidential resources. Although sometimes provably secure [11], these systems restrict third-party code to subsets of HTML and JS to enable analysis.

More generally, all data protection mechanisms discussed above implement some form of one-time *access control* on data available to third-party code. As such, they are completely ineffective when the third-party code legitimately needs confidential data to provide functionality, but must be prevented from disclosing it in unexpected ways. *Information flow control* (IFC) within the web browser is an obvious, holistic method to solve this problem. With IFC, the browser can allow third-party code to access confidential data, but monitor flows of information within the third-party code (either finely or coarsely) and selectively disallow unexpected flows, thus supporting both functionality and security. Unsurprisingly, a number of solutions based on IFC have been proposed for web browsers [12], [13], [14], [15], [16], [17], [18], [19]. However, all these solutions have two significant shortcomings — they either do not handle or abstract over the event handling logic of the browser and they do not handle the browser APIs completely. In this paper, we design, formalize, prove sound and implement an IFC solution that addresses these shortcomings.

Shortcoming 1: Event handling logic. Existing IFC solutions for web browsers do not treat the browser's *event handling* logic completely and accurately. A webpage is reactive: Input events like mouse clicks, key presses and network receives trigger JS functions called *handlers*. The event handling logic of a browser is complex. Each input event can trigger handlers registered not just on the node on which the event occurs (e.g., the button which is clicked), but also on its ancestors in the HTML parse tree. This can leak information implicitly through the presence or absence of links between the node and its ancestors. However, existing work on IFC for web browsers either does not consider the reactive nature of webpages at all (focusing, instead, only on the sequential JS code within a handler) [14], [15], [13], [20], [16] or it abstracts away from the details of the event handling logic, thus side-lining the problem [17], [18].

In contrast, in this work we (a) Demonstrate through working examples that information leaks through the event loop logic are real (and subtle) and, thus, should not be abstracted, (b) Enrich a formal model of the event loop of a browser with fine-grained IFC to prevent such leaks, (c) Prove that our proposed extension is sound by establishing noninterference, a standard property of IFC, and (d) Implement our IFC solution in WebKit, an in-production browser engine used in many browsers (including Apple's Safari). Our IFC-enriched model of the event loops and the noninterference proof are parametric in the sequential small-step semantics and IFC checks of individual event handlers. Additionally, our solution can be layered over any existing label-based, fine-grained IFC solution for sequential JS that satisfies noninterference, e.g., [14], [15]. To test and evaluate the cost of our IFC checks, we extend Bichhawat *et al.*'s IFC instrumentation for individual handlers in WebKit's JS bytecode interpreter [15].

As a further contribution, we observe empirically that event handlers in web browsers do *not* necessarily execute atomically. *Every* existing work on IFC in web browsers (and beyond) incorrectly assumes the opposite. Chrome, Firefox and Internet Explorer sometimes temporarily suspend an event handler at specific API calls to handle other waiting events. The suspended handlers resume after the other waiting events have been handled. This behavior can be nested. As we show through examples, this kind of preemption can also cause implicit information leaks. We model this preemption in our formalism and our IFC instrumentation and implementation prevent leaks through preemption. This adds complexity to our model: We cannot model the state of the event loop with just one call stack for the current event (as existing work does). Instead, we model the state of the event loop with a stack of call stacks — the topmost call stack is the state of the current event and the remaining call stacks are the states of previously suspended events. We note that in the future, web browsers are expected to aggressively support cooperative yielding and resumption of threads with multiple event loops (this is anticipated by the HTML5 specification [21]); our IFC solution should provide a stepping stone to such general settings.

Shortcoming 2: Browser APIs. Existing IFC solutions for web browsers do not cover all APIs of the DOM specification [22]. The Document Object Model or DOM is the parsed, internal state of a webpage that includes all visible and invisible content, embedded handlers and scripts, JS primitive functions and global browser information. The DOM can be read and modified from JS through many native, standard APIs provided by every browser. These APIs, called the DOM APIs, were introduced into browsers in three stages, now dubbed DOM Levels 1, 2 and 3. The DOM is *the* main shared state (memory or heap) for JS code executing in the browser. Its APIs often have complex implementations and, hence, any IFC solution *should* carefully instrument these APIs for IFC and account for that instrumentation in the soundness proof. However, existing IFC solutions for web browsers either completely ignore the DOM [14], [13], [15] (and consider a JS core with a standard heap), or instrument only a part of the DOM [20], [18], [17], [23], [24]. Other work does not specify how far the DOM was instrumented and does not prevent all leaks [16]. Formal models of the DOM outside of IFC are limited — we know of only two and both are partial [25], [26].

In our work, we model all DOM APIs up to and including Level 3, and instrument them (in WebKit) to track fine-grained taints and enforce IFC. This is nontrivial because we had to consult the implementation of the DOM APIs in WebKit to resolve ambiguities in the standard DOM specification [22]. For instance, in the case of the API `getElementById('id')`, which is supposed to retrieve a unique element `id`, the specification does not specify behavior when several elements have the same `id`. To resolve such ambiguities, we turn to WebKit's implementation. In doing so, we also found a new set of leaks which arise due to optimizations in WebKit. Our model of all DOM APIs can be added to any prior sequential model of JS in the form of extra primitive JS functions. Our noninterference proof (for the event loop) also carefully analyzes our IFC instrumentation of every DOM API and shows that our design prevents information leaks. Our model of the DOM, which may be of interest even outside of IFC, is formalized as (type-checked) OCaml code and is available online from the authors' homepages. We do not describe the DOM API or our instrumentation of it in any detail in this paper, except through examples (we focus on the conceptually harder event loop in the technical sections of this paper).

Summary of contributions. To the best of our knowledge, this is the first web browser IFC work that handles event loops and the DOM up to Level 3. To summarize, we make the following contributions.

- We formalize the event handling loop of a browser, highlighting how it can leak secrets implicitly, and develop a fine-grained dynamic IFC monitor for it.
- We develop a formal model of the DOM up to Level 3. The model is abstracted from the DOM specification and its implementation in an actual browser engine (WebKit). We enrich our model with provisions for fine-grained IFC.
- We prove a form of reactive noninterference for a termination-insensitive attacker. Our proofs are parametric on preemption points and a provably sound IFC monitor for sequential JavaScript.
- We implement these concepts in a fully-functional browser (Apple's Safari, based on WebKit) and observe moderate performance overhead.

II. Background

A. Information Flow Control

Information flow control (IFC) refers to controlling the flow of (confidential) information through a program based on a given security policy. Typically, pieces of information are classified into security labels and the policy is a lattice over labels. Information is only allowed to flow up the lattice. For illustration purposes often the smallest non-trivial lattice $L < H$ is used, which specifies that public (low, L) data must not be influenced by confidential (high, H) data. In our instrumentation labels are drawn from a product lattice where each dimension represents a unique web domain. IFC can be used to provide confidentiality (or integrity) of secret (trusted) information. We are only interested in confidentiality here.

In general, information can flow along many channels. Here, we consider *explicit* and *implicit* flows. Covert channels like timing or resource usage are beyond the scope of this work. An explicit flow occurs as a result of direct assignment, e.g., the statement `public = secret + 1` causes an explicit flow from `secret` to `public`. An implicit flow occurs due to the control structure of the program. For instance, in the program `public = false; if (secret) public = true`, the final value of `public` is equal to the value of `secret` even though there is no direct assignment mentioning both `secret` and `public`. Leaking a bit like this can be magnified into leaking a bigger secret bit-by-bit [27].

Research has considered static methods such as type checking and program analysis, which verify the security policy at compile time [28], [29], [30], [31], dynamic methods such as black-box approaches as well as attaching secrecy labels to runtime values and tracking them through program execution [32], [33], [14], [34], [35], [36], and hybrid approaches that combine both static and dynamic analyses to add precision to the analysis [37], [38], [39], [15] for handling the leaks described above. The correctness of these approaches is often stated in terms of a well-defined property known as *noninterference* [40], which basically stipulates that high input of a program must not influence its low output. While noninterference is too strong a property in practice, it is a useful soundness check for IFC mechanisms.

We are interested in IFC through runtime monitoring with labels attached to all values. Preventing explicit flows that violate noninterference is trivial via runtime monitoring, once all values in the system are labeled. However, it can be difficult to prevent leaks due to implicit flows. Dynamic IFC approaches usually use a notion of context label ($\mathcal{PC}$), which represents an upper bound on the labels of all the values that have influenced the control flow at the current instruction, and join this label with the label derived from explicit flow for every variable assignment. However, it can be shown that this is not sufficient for noninterference [41] when labels attached to variables may change over time, as even assignments in code that is *not* executed may lead to implicit flows. Listing 1 illustrates unexecuted branches that may leak information. This code snippet effectively copies the (secret) value of `h` into the public variable `l` via another

```
1 l = false, t = false
2 if (h == false)
3   t = true
4 if (t != true)
5   l = true
```

Listing 1: Example for implicit flow

public variable `t` without `l` being labeled secret. This is because *either* the first condition is true or the second but never both. The *no-sensitive-upgrade* (NSU) check [42], [32] rejects such programs by prohibiting modification of a public variable in a secret context (when the $\mathcal{PC}$ label is high), terminating the program if it tries to do so. In this program, when `h` is `false`, the assignment on line 3 is forbidden and the program is terminated. Programs executed under NSU satisfy the soundness property *termination-insensitive non-interference* [29]. Intuitively, this soundness criterion requires the absence of information leaks for an attacker who cannot observe termination of programs (a formal definition is given in Section IV).

B. Document Object Model and Event Handling

Document Object Model: The *document object model* (DOM) is the parsed, internal state of a webpage that includes all visible and invisible content, embedded handlers and scripts, JS primitive functions and global browser information. It can be accessed from JS programs via the DOM API, which provides interfaces for JS programs to read, modify, create and delete parts of its state. DOM API calls have been added to browsers gradually in stages that are dubbed levels. The current standard implemented in most browsers is Level 3 (which subsumes Levels 1 and 2). The DOM graph, which represents the visual content described by HTML, can be navigated in various directions using API calls, e.g., from a node to its parent, to its first and last children, to its left and right siblings, etc.

Nodes of the DOM graph have various types, like an element node, a text node, a document fragment and many others. All of these are well-defined data structures in the DOM specification [22]. A special kind of data structure of significance to us is the *live collection*. It is returned by some DOM search APIs. A live collection always represents the current state of the DOM graph, i.e., changes in the DOM graph are reflected in future uses of these collections. As an example, consider the function call `document.getElementsByTagName('div')`. This calls returns a reference to a list containing all elements (element nodes) that have the tag name `div`. If another node with tag name `div` is added to the DOM graph after the call, it will be present in subsequent uses of the list. Similarly, if an element with tag name `div` is removed from the DOM graph, this element is removed from the list automatically.

Event handling: Web pages may be reactive. They can respond to *events* like mouse clicks, network responses and key presses by invoking *event handlers*, which are JS functions. Every event has a *target*, a node in the DOM graph, where the event originates (e.g., if the mouse is clicked on a button, then the button would be the target of the resulting `click` event). Event handlers for specific events can be associated with every node programmatically through the browser API `node.addEventListener(event, handler, boolean)`. Every handler is registered with one of the following attributes: *target and bubble* or *capture and target* by setting the `boolean` third argument to `false` and `true`, respectively. The meaning of these attributes is explained below.

The event loop of a browser is complex. Browsers maintain a list of incoming, pending events called the *event queue*. Events in the queue are processed one at a time (not necessarily in FIFO order). The processing of an event is called a *dispatch*. When an event is dispatched, the handlers registered for the event associated with the event's target node are executed. Additionally, certain handlers registered for the event associated with the nodes on the entire path from the target to the root of the DOM graph are also executed. This path is called the *propagation path*. To dispatch an event, first, this propagation path is computed by traversing the DOM graph. This path remains fixed during event dispatch even though the shape of the DOM graph may change due to the execution of handlers. Next, the handlers are executed in three phases:

- The *capture phase* executes all capture and target handlers associated with all nodes from the root to the target's parent, starting from the root.
- The *target phase* executes all the handlers associated with the target.
- The *bubble phase* executes all target and bubble handlers associated with all nodes from the target's parent to the root, starting from the target's parent.

Finally, the browser may execute *default actions* (the browser's in-built actions) associated with the event. For example, middle-clicking a url in Chrome opens the url in a new tab.

Events can be dispatched by external actions (like a physical mouse click) or programmatically using the API call `dispatchEvent`(). When an event is dispatched programmatically, default actions are usually not executed. An exception to this is the `click` event.

Every dispatched event has three flags which can be set by any executing event handler through API calls to modify the execution of subsequent handlers. The flag `stopImmediatePropagation` terminates handling of the event immediately after the current handler ends. No other handlers are executed, but default actions are executed. The flag `stopPropagation` is similar but it terminates handling after all handlers associated with the current node have executed. The flag `defaultPrevented` prevents the execution of default actions.

```
var p = document.getElementById('para');
p.onclick = function() {
   alert('In click');
   p.innerHTML += 'click';
};
window.onresize = function() {
   p.innerHTML = 'resize ';
};
```

Listing 2: Preemption in browsers

Handler preemption: An event handler can be paused or suspended at specific API calls like `alert()` or `confirm()` that wait for user response. When suspended at these APIs, some browsers choose to dispatch other events in the event queue, which makes the execution of JS in these browsers resemble cooperative scheduling. Consider the snippet in Listing 2. Assume that the page has a paragraph element with id `para`, which is bound to the variable `p`. The script registers two handlers: one for the `onclick` (click) event on `p` and the other for the `onresize` (resize) event on the global object, `window`. The user clicks the paragraph `p`, which displays an `alert` dialog box. Before dismissing the dialog box, the user resizes the main window thereby registering an `onresize` event. In some browsers, the `onresize` handler will execute while the `onclick` handler is still suspended. This will cause the word `resize` to appear in the paragraph `p` before the word `click`. (We verified this behavior on Chrome version 40.0.2214.111.) On other browsers, resizing the window will not be allowed until the alert dialog box is dismissed or the `onresize` event will not be dispatched until the dialog box is dismissed and `onclick` has finished execution. To account for this browser-dependent behavior, we parametrize our model with a set of preemption points — the API calls at which handler execution may be preempted.

III. Overview of Challenges and Approach

In this section, we highlight some possible information leaks in client-side JS due to subtleties of handler preemption, the event loop logic, the DOM and browser optimizations, and explain at a high-level how our work prevents these leaks. But, first, we explain our attacker model.

Approach and attacker model. We perform fine-grained, flow-sensitive taint tracking in the DOM (and through the event loop), so we assume here onward that taints are attached to individual fields of DOM elements. Thus, the parent, first and last child, and left and right sibling pointers of a DOM node can have independent taints and the content of the node can have yet another taint. This is quite standard in fine-grained dynamic taint tracking [14], [15]. Conceptually, taints may be drawn from an arbitrary security lattice;

```
1   function foo() {
2     ...
3     pub = true;
4     if (sec)
5       preemption-point
6     ...
7     pub = false;
8   }
9
10  function bar() {
11    conf = pub;
12  }
```
Listing 3: Implicit leak via preemption

our prototype implementation uses a subset lattice where the taint on a field is an upper bound on the set of web domains which may have influenced the field. We also attach labels to individual events and event handlers. A value v labeled with label ℓ is written v^{ℓ}.

Our attacker resides at an arbitrary security level $\mathcal{A}$ of the lattice. We assume that the attacker can observe all references and values $\ell \leq \mathcal{A}$ that are reachable from the global object (`window` object) of the browser. The attacker cannot directly observe internal data structures like the call-stack, event queues, etc. We limit attention to a termination-insensitive attacker. Leaks due to termination, timing, progress and other side channels are outside our threat model.

Examples Through a series of examples, we demonstrate subtle implicit leaks through handler preemption, event loops, the DOM APIs and browser optimizations. These pose a challenge for building a dynamic flow-sensitive IFC monitor. For simplicity, our examples demonstrate only one-bit leaks. In all our examples, we assume the lattice $L < H$ (low $<$ high) and an attacker at level L.

Handler preemption: Our first example highlights a possible leak due to preemption in the middle of a handler. As noted earlier, every prior work on IFC for web browsers (and reactive systems in general) ignores such preemption and, hence, will miss this attack. Listing 3 describes two handlers, `foo` and `bar`, for two arbitrary events e_1 and e_2, respectively. We assume that both events are in the event queue already and that the handler for e_1, i.e., `foo` is currently executing, while the other handler has not yet started executing. In the function `foo`, we initialize a variable `pub` to `true` and branch on a H-labelled variable `sec` to decide whether or not to execute an instruction at line 5, which is a preemption point according to the browser semantics. If `sec` is `true`, then at line 5, `foo` will be suspended and `bar` will start executing while `pub` is `true`. From the code of `bar`, it is clear that the variable `conf` will end up with the value `true`. If, on the other hand, `sec` is `false`, then the preemption point is never executed, so `bar` executes after `foo` sets `pub` to `false` and `conf` ends with `false`. Hence, in a browser that preempts at line 5, this program implicitly copies `sec` to `conf`.

The relevant question is how we may prevent this leak with dynamic IFC. A naive solution, based on handling of implicit flows in sequential programs, would be to carry the $\mathcal{PC}$ from `foo` to `bar` at line 5, i.e., to execute `bar` with $\mathcal{PC} = H$ if line 5 executes. Although intuitive, this naive solution is *unsound*: It prevents the leak only if `conf` is initially labeled L, but still allows the leak if `conf` is initially labeled H. If `conf` is initially labeled L, when `sec` is $\texttt{true}^H$, `bar` executes with $\mathcal{PC} = H$ and this prevents the assignment $\texttt{conf}^L = \texttt{pub}$ due to the no-sensitive-upgrade (NSU) check. Hence, the program does not leak (to a termination-insensitive attacker). However, if `conf` is initially labeled H, this assignment is allowed, so the program leaks information. More specifically, if `conf` is initially labeled H, then `conf` ends with value $\texttt{true}^H$ if `sec` is $\texttt{true}^H$, and with value $\texttt{false}^L$ if `sec` is $\texttt{false}^H$. The final values of `conf` — $\texttt{true}^H$ and $\texttt{false}^L$ — are distinguishable for the attacker. So, $\texttt{sec}^H$ is leaked by the program.

To solve this problem, we attach minimum $\mathcal{PC}$ labels to all handlers and execute a handler at a preemption point only when the handler's $\mathcal{PC}$ label is at least as high as the $\mathcal{PC}$ of the context. In this example, `bar` must have a label at least H in order to execute after line 5. This minimum label ensures that after `bar` executes, `conf` has label H irrespective of earlier code paths so, trivially, there are no information leaks. Note that we do *not* specify labels for handlers through additional program annotations. Instead, a handler inherits the label of the context in which the handler was associated with the event. This is essential for soundness, else the handler's execution may implicitly leak information from the context in which the handler was associated with the event.

Event phases: As described in Section II, event handling happens in several phases and the handlers executed in response to an event depend on the shape of the DOM, which is itself dynamic. This can result in implicit information leaks unless the IFC monitor is carefully designed. The example in Listing 4 illustrates this point. We have a low variable `pub` initially set to `false`, and two nodes `a` and `b`. The `onClick` handlers (the handlers invoked when the mouse is clicked on the node) for `a` and `b` are set to `aCk` and `bCk`, respectively. `aCk` is a capture and target phase handler (third argument to `addEventListener()` is `true` on line 2) and `bCk` is a target and bubble phase handler (third argument to `addEventListener()` is `false` on line 3). Based on a high variable `sec`, we either make `b` a child of `a` or we do not. Then, on line 6, a mouse click is programmatically dispatched on `b`. `b` is the target of the event, so `bCk` will definitely execute. However, if `sec` is `true`, then `aCk` will also execute (before `bCk`) because `aCk` is registered as a capture phase handler and `a` is the

```
pub = false
a.addEventListener ("click", aCk, true)
b.addEventListener ("click", bCk, false)
if (sec)
  a.appendChild (b)
b.dispatchEvent ("click")

function aCk () {
  pub = true
}
function bCk () { ... }
```

Listing 4: Example leak due to lack of *propagation path labels* across event phases

```
1  nodes =
2     document.getElementsByTagName("div")
3  x = nodes.length
4  newNode = document.createElement("div")
5  if (sec)
6    document.body.appendChild(newNode)
7  y = nodes.length
```

Listing 5: Example leak via live collections

parent of b. The execution of aCk will change pub to true. Hence, sec is copied to pub implicitly.

To prevent this leak, let us examine why the leak happens. Essentially, depending on the value of sec, either there exists a link between a and b or there does not. This link then determines whether or not aCk executes. Hence, the leak can be prevented by ensuring two things: (1) The link between a and b is labeled with the $\mathcal{PC}$ when the link is formed (here, the label would be H) and, (2) When a handler tied to a node such as aCk is executed, the execution's $\mathcal{PC}$ is at least as high as the labels of all links on the path from the target node to this node (in this case, the link from a to b, which would be labelled H). In our DOM instrumentation, (1) happens for free because links between nodes are ordinary pointer fields, and all fields inherit the $\mathcal{PC}$ label of the context upon update. On the other hand, (2) requires special care: Our instrumentation executes all handlers for an event with a $\mathcal{PC}$ at least as high as the *propagation path label*, which is the join of the labels on the pointers in the propagation path of the event's target. This subsumes (2) and also prevents similar implicit leaks.

Live collections: Some DOM API functions that search the DOM graph return a live collection of nodes. A live collection is automatically updated as nodes that match the search criteria are added or removed from the DOM graph. This can leak information implicitly. Listing 5 shows a simple example. The variable nodes is bound to the live collection of all nodes in the DOM that have the tag div. The length (size) of this collection is snapshotted in the variable x. Depending on the value of a high variable sec, either a new node newNode with tag div is added to the DOM or it is not. If the node is added, this will automatically (and implicitly) update the live collection. The new length of the live collection is snapshotted in y. If y and x are different, then sec is true, else it is false. This is an implicit leak of sec to x and y.

Two recent proposals [24], [20] offer similar solutions to this problem: They require the programmer to provide a security label for every tag like div. The IFC monitor ensures that this label is an upper bound on the $\mathcal{PC}$ of the context in which any node with that tag is added to the DOM. Any value computed from the live data structure, e.g., length, inherits this label. In our example, the tag div must have the label high in order for line 7 to execute. So, nodes.length, x and y would all be labelled high, preventing the leak.

Although elegant, this solution requires the programmer to specify labels for each tag. Our approach, on the other hand, does not require any assistance from the programmer. We rely on the observation that browsers implement properties of live data structures like length by traversing the DOM graph.[1] Consequently, simply applying our usual IFC checks to the implementations of methods of live data structures yields sound enforcement. In our example, if line 6 succeeds, then due to the NSU check on pointer update, the *last-child* pointer of document.body must already be labeled high. Hence, irrespective of whether or not newNode is added, traversing the DOM graph to compute the length field will always return a high value. Hence, x and y will be labelled high, preventing any leak. Note that our solution requires fine-grained labels on all fields in the DOM (which we have). Prior work mentioned above did not include a full instrumentation of the DOM and, hence, could not have adopted our solution.

Browser optimizations: Browsers often create auxiliary internal data structures to speed up commonly invoked DOM API functions. Any IFC instrumentation of the DOM must also label these auxiliary data structures. This can be subtle as we illustrate here. WebKit, the browser engine we instrument, maintains a HashMap from node ids to DOM nodes to speed up the very common lookup function getElementById('iden'), which returns the DOM node with id iden. Whenever a node is inserted into the DOM (as the child of another already present node), the node is also added to this HashMap. However, for various reasons, this map contains a *subset* of all nodes in the DOM. If a certain id does not exist in the HashMap, then the DOM must be traversed to search. Clearly, to prevent an implicit leak, it is essential that the $\mathcal{PC}$ of the context in which a node is added to the DOM also labels the entry of the node in the HashMap (if the entry exists). However, it turns out

[1]A live data structure is marked invalid as soon as there is any change to the DOM. A subsequent method call on the data structure results in a fresh DOM traversal.

```
c.setAttribute("id","ifc")
if (sec)
  b.appendChild(c)
a.appendChild(b)
x = document.getElementById("ifc")
```

Listing 6: Example leak showing insufficiency of $\mathcal{PC}$ for preventing leak via getElementById

that just this labeling is *insufficient* in cases where the node being added to the DOM has children.

As an illustration, consider the example of Listing 6. Three nodes `a`, `b` and `c` have been created before the program starts, but only `a` is already attached to the DOM. Assume that `b` is a high node, i.e., its own label and its fields are all labeled H. The program assigns `c` the id `ifc`, which will be searched for later. Based on the value of a high variable `sec`, `c` is either made a child of `b` or not. Then, `b` is made a child of `a`. This adds `b` and possibly `c` to the DOM. Since this instruction executes with $\mathcal{PC} = L$, following the labeling rule described above, `c`'s entry in the HashMap, if any, will be labeled L. The last line of the program searches for the id `ifc` in the DOM and stores the result in variable `x`. At the end of the program, `x` will be $\texttt{null}^H$ if `sec` is $\texttt{false}^H$ (note that the DOM traversal to discover that a node with id `ifc` does not exist will pass through `b`, which is labelled H and, hence, will return $\texttt{null}^H$). On the other hand, `x` will be $\texttt{c}^L$ if `sec` is $\texttt{true}^H$ due to a successful HashMap lookup. Since $\texttt{null}^H$ and $\texttt{c}^L$ are distinguishable for the adversary, this program leaks `sec` into `x`.

This leak happens because, when `sec` is `true`, we completely ignore the H label of the link between `b` and `c` when labeling `c` in the HashMap. The correct rule for labeling the HashMap on insertion is that when a node, say `b`, is added to the DOM graph, any node accessible from `b` must be added to the HashMap with a label that is equal to the *join of all labels* on the path from `b` to that node. With this rule in place, when `c` is inserted into the HashMap, its label is H, not L, and this prevents the leak.

IV. Information Flow Control for the DOM and the Event Loop

Our central technical contribution is IFC-enhanced models of the DOM Core Level 3 [22] and of the event loop of a web browser, including event preemption. Our models plugs into any model of sequential JS execution within a handler if the sequential model uses dynamic fine-grained taints for IFC. Our model of the DOM extends such a sequential model with additional primitive functions (the DOM API) and our model of event loops provides a wrapper around such a model, resulting in a reactive system. To prove noninterference, we assume that the IFC on sequential JS is noninterfering and satisfies some specific lemmas (like the standard confinement lemma). The specific sequential JS model we use is taken from [15].

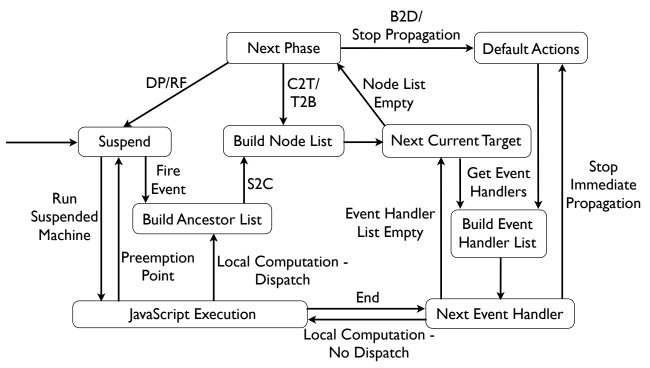

Figure 1: State machine description of an event loop

IFC for the DOM. As described in Section II-B and illustrated in Section III, the DOM is a graph whose nodes correspond to various elements of a web page. Technically, each node is a JS object. The standard rules of IFC from the sequential JS apply to these objects as well. In particular, every field of every node, including every pointer to the node's children, siblings and parent, is given a separate security label. This label is always at least as high as the $\mathcal{PC}$ of the context in which the field was last modified.

Separately, we build an OCaml model of all DOM API functions. We follow the DOM specification [22] and refer to its implementation in WebKit (our target browser engine) to resolve ambiguities and to add additional data structures like the HashMap of Section III(d), which are not part of the specification. Finally, we instrument the API functions with IFC checks. We follow the NSU rule to prevent implicit flows. Two interesting aspects of the model were highlighted in Section III(c) and (d). We do not describe this model in any further detail here. Instead, in the rest of this section, we focus on our model of a browser's event loop and its IFC enhancement.

A. The event loop and its IFC enhancement

As explained in Section II, the event loop is a transition system that collects events and dispatches them to handlers through a reasonably complex logic. Additionally, event handlers can be preempted at browser-specific API calls. Fig. 1 describes the event loop as an abstract state machine. The machine starts in a suspended state (and returns to it when there are no events; this transition is not shown). The occurrence of an event (called "Fire event") transitions the machine to what we call the *start phase*. In this phase, the propagation path, or the list of nodes from the target of the event to the root of the DOM graph, is computed. This list is fixed for the duration of the event's dispatch. Then, the machine transitions through the capture, target and bubble phases described in Section II. In Fig. 1, the transitions are labelled S2C (start to capture), C2T (capture to target), T2B (target to bubble), B2D (bubble to default). The last phase of execution is what we call the default phase, where the default actions are executed.[2] The execution of some handlers can be bypassed using the flags `stopPropagation` and `stopImmediatePropagation`. When either the default actions are prevented (transition DP), an event has been handled completely (transition RF), or a preemption point is reached, the machine enters the suspend state, where it can dispatch other events.

Concretely, we model a state of the machine as a pair $\kappa = \langle \Sigma, \theta \rangle$, where Σ is a stack of frames and θ is a shared heap (which includes the DOM graph). Each frame corresponds to one event which has been dispatched, but whose handling has not yet completed. The top most frame corresponds to the event currently being handled; every other frame has been suspended (prior to completion) to handle events in frames above it. The stack frame is a rich data structure that encodes the entire state of an event's dispatch including the state of the currently running handler, phase information and handlers that have not yet run. The stack frame is a tuple $\nu = \langle C, M, \ell, E, N, \mathcal{A}, \mathcal{N}, \mathcal{H}, \mathcal{P} \rangle$, where:

- C : While a handler is active, C is a configuration of the sequential JS machine, minus the heap, which is shared. It has the form $C = \langle \iota, \sigma, \rho \rangle$, where ι is a pointer to the instruction (in the heap) to be executed next, σ is the call stack of the current handler, and ρ is the $\mathcal{PC}$-stack for the current handler. We write $\iota(C)$, $\sigma(C)$, and $\rho(C)$ to denote the current instruction, call-stack and $\mathcal{PC}$-stack of the configuration C. Between two phases or between two handlers, when no handler is active, C temporarily takes the form ϕ.
- M : Either S (suspended) or R (running). All frames in Σ except the topmost are always in the S state. When the topmost frame is in state S, a new event can be dispatched by adding a new frame on top.
- ℓ : The security label in which the frame ν was created. This indicates the minimum level at which instructions execute in that frame. The function $\Gamma(\nu)$ returns the value of ℓ in frame ν.
- E : A pointer to the event object for which the frame ν was created.
- N : A pointer to the target object for the event E.
- $\mathcal{A}$: A list of pointers to nodes in the propagation path of the target. This list starts with the target's parent node and ends at the root of the DOM. The list is fixed when the frame is created and does not change afterward.
- $\mathcal{N}$: A list of pointers to nodes on which handlers have yet to be run in the current phase. This list is re-populated at every phase transition and gets smaller as the phase progresses.
- $\mathcal{H}$: A list of pointers to event handlers (functions) at the current node that are yet to be executed.
- $\mathcal{P}$: The current phase of execution for the event. It can be $\mathcal{S}$, $\mathcal{C}$, $\mathcal{T}$, $\mathcal{B}$ or $\mathcal{D}$ for start phase, capture phase, target phase, bubble phase and default phase, respectively.

The transitions of this state machine are described as small step operational semantics in Figure 2. The transition relation has the form $\langle \Sigma, \theta \rangle \twoheadrightarrow_{\alpha} \langle \Sigma', \theta' \rangle$ where $\alpha = \cdot | (e, o)$ is a label. $\alpha = (e, o)$ labels the firing of event e on target object o. All other transitions are labeled $\alpha = \cdot$. The top frame of a stack S is denoted $!S$. Some of the rules make use of meta functions like `getPathLabel`, which we describe informally as we explain the rules; all of these functions have a formal description in our OCaml model. We describe the rules of Figure 2 below.

`Local Computation-No Dispatch`: When the current instruction in the top frame of Σ (i.e., $\iota(!\Sigma.C)$) is neither *dispatchEvent* (programmatic event dispatch) nor a preemption point, then we can execute this instruction using the small-step semantics of sequential JS. The relation $\theta, C \rightarrow \theta', C'$ is the small-step transition relation of the underlying sequential JS machine.

`Local Computation-Dispatch`: This rule executes the programmatic dispatch call, *dispatchEvent*. We push a new frame on Σ. The current frame's state changes to S (suspended). The function `getPathLabel` joins the labels of the nodes on the propagation path of the event target `o`. The function `getAncestors` returns the propagation path of `o`. We set the label ℓ in the new top frame to the join of the current $\mathcal{PC}$ and the path label, as explained in Section III(b). The current phase of the new frame is set to the start phase, $\mathcal{S}$.

`Preemption-Point`: If the current instruction is a preemption point according to the browser semantics (captured in the browser-dependent check $\mathit{isPreemptionPoint}(\iota(C_m)) = \mathit{true}$), then this rule changes the state of the topmost frame of Σ to S. This allows another event to fire through the rule `Fire Event` (which applies only when the topmost frame has state S).

`Run Suspended Machine`: If the frame on the top of the stack Σ is currently suspended, then it may be resumed by setting the state to R.

`End`: When the current handler has finished executing ($\iota(C_m) = \mathit{end}$), this rule sets the configuration C in the top frame to ϕ. This allows the machine to transition phase or execute the next handler.

`Fire Event`: This rule fires a queued (non-programmatic) event `e` on the target node `o`. The rule is mostly similar to `Local Computation-Dispatch`, but additionally prevents the firing of a low event in a high context, to handle the problem described in Section III(a).

`Start-to-Capture`: This rule transitions from the start to the capture phase. The pending node list $\mathcal{N}$ is initialized to nodes which must be traversed during the

[2]The words "start phase" and "default phase" are not taken verbatim from the event specification [43]. We use these terms as they are similar to capture phase, target phase and bubble phase, which the specification mentions.

Local Computation-No Dispatch

$$\frac{\begin{array}{c} !\Sigma.C = C_m \qquad !\Sigma.M = R \qquad \theta, C_m \rightarrow \theta', C'_m \\ \mathit{isPreemptionPoint}(\iota(C_m)) = \mathit{false} \\ \iota(C_m) \neq o.\mathit{dispatchEvent}(e) \\ \Sigma' := \Sigma[!\Sigma.C := C'_m] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta'}$$

Local Computation-Dispatch

$$\frac{\begin{array}{c} !\Sigma.C = C_m \qquad !\Sigma.M = R \qquad \theta, C_m \rightarrow \theta, C'_m \\ \iota(C_m) = o.\mathit{dispatchEvent}(e) \\ \Sigma'' := \Sigma[!\Sigma.C := C'_m, !\Sigma.M := S] \\ e.\mathit{eventType} \neq \text{“}\texttt{click}\text{”} \implies e.\mathit{defaultPrevented} := \mathit{true} \\ \ell_p = \mathit{getPathLabel}(o, \theta) \qquad \mathcal{A} = \mathit{getAncestors}(o, \theta) \\ \ell := \Gamma(!\rho(C_m)) \sqcup \ell_p \\ \Sigma' := \langle \phi, S, \ell, e, o, \mathcal{A}, [], [], \mathcal{S} \rangle :: \Sigma'' \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Preemption-Point

$$\frac{\begin{array}{c} !\Sigma.C = C_m \qquad !\Sigma.M = R \qquad \theta, C_m \rightarrow \theta', C'_m \\ \mathit{isPreemptionPoint}(\iota(C_m)) = \mathit{true} \\ \Sigma' := \Sigma[!\Sigma.C := C'_m, !\Sigma.M := S] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta'}$$

Run Suspended Machine

$$\frac{!\Sigma.C = C_m \quad !\Sigma.M = S \quad \Sigma' := \Sigma[!\Sigma.M = R]}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

End

$$\frac{\begin{array}{c} !\Sigma.C = C_m \qquad !\Sigma.M = R \qquad \iota(C_m) = \mathit{end} \\ \Sigma' := \Sigma[!\Sigma.C := \phi] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Fire Event

$$\frac{\begin{array}{c} !\Sigma.M = S \qquad \ell_p = \mathit{getPathLabel}(o, \theta) \\ \mathcal{A} = \mathit{getAncestors}(o, \theta) \quad \ell := \ell_p \sqcup \Gamma(\theta(e)) \sqcup \Gamma(\theta(o)) \\ \Big(!\Sigma.C = \phi \wedge \ell \geq \Gamma(!\Sigma)\Big) \vee \Big(\ell \geq \Gamma(!\rho(!\Sigma.C))\Big) \\ \Sigma' := \langle \phi, S, \ell, e, o, \mathcal{A}, [], [], \mathcal{S} \rangle :: \Sigma \end{array}}{\Sigma, \theta \twoheadrightarrow_{(e,o)} \Sigma', \theta}$$

Start-to-Capture

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{P} = \mathcal{S} \\ \mathit{nl} = \mathit{reverse}(!\Sigma.\mathcal{A}) \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := \mathit{nl}, !\Sigma.\mathcal{P} = \mathcal{C}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Capture-to-Target

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{P} = \mathcal{C} \\ !\Sigma.\mathcal{N} = [] \qquad !\Sigma.\mathcal{H} = [] \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := [!\Sigma.N], !\Sigma.\mathcal{P} = \mathcal{T}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Target-to-Bubble

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{P} = \mathcal{T} \\ !\Sigma.\mathcal{N} = [] \qquad !\Sigma.\mathcal{H} = [] \\ !\Sigma.E.\mathit{bubbles} = \mathit{true} \implies !\Sigma.\mathcal{N} := !\Sigma.\mathcal{A} \\ \Sigma' := \Sigma[!\Sigma.\mathcal{P} = \mathcal{B}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Bubble-to-Default

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{P} = \mathcal{B} \qquad !\Sigma.\mathcal{N} = [] \\ !\Sigma.\mathcal{H} = [] \qquad !\Sigma.E.\mathit{defaultPrevented} = \mathit{false} \\ \mathit{nl} = \mathit{getDefaults}(!\Sigma.\mathcal{A}, !\Sigma.N, !\Sigma.E.\mathit{bubbles}) \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := \mathit{nl}, !\Sigma.\mathcal{P} = \mathcal{D}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Default Prevented

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{P} = \mathcal{B} \qquad !\Sigma.\mathcal{N} = [] \\ !\Sigma.\mathcal{H} = [] \qquad !\Sigma.E.\mathit{defaultPrevented} = \mathit{true} \\ \Sigma' := \Sigma[!\Sigma.\mathcal{P} = \mathcal{D}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Remove Frame

$$\frac{\begin{array}{c} !\Sigma.C = \phi \quad !\Sigma.\mathcal{P} = \mathcal{D} \quad !\Sigma.\mathcal{N} = [] \quad !\Sigma.\mathcal{H} = [] \\ \Sigma' := \Sigma - !\Sigma \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Get Event Handlers

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{N} = n :: \mathcal{N}' \qquad !\Sigma.\mathcal{H} = [] \\ !\Sigma.E.\mathit{stopPropagation} = \mathit{false} \\ \mathit{ehl} := \mathit{getEventHandlers}(!\Sigma.E, n, !\Sigma.\mathcal{P}, \theta) \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := \mathcal{N}', !\Sigma.\mathcal{H} := \mathit{ehl}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Stop Propagation

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{H} = [] \\ !\Sigma.E.\mathit{stopPropagation} = \mathit{true} \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := [], !\Sigma.\mathcal{P} := \mathcal{B}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Run Event Handlers

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.\mathcal{H} = (\ell_c, f_m) :: \mathcal{H}' \\ !\Sigma.E.\mathit{stopImmediatePropagation} = \mathit{false} \\ \rho(C_m) := [!\Sigma.\ell \sqcup \ell_c \sqcup \Gamma(f_m)] \\ \iota(C_m) := \mathit{null} \qquad \sigma(C_m) = \mathit{emptyCallFrame} \\ \Sigma' := \Sigma[!\Sigma.C := C_m, !\Sigma.\mathcal{H} := \mathcal{H}'] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Stop Immediate Propagation

$$\frac{\begin{array}{c} !\Sigma.C = \phi \qquad !\Sigma.E.\mathit{stopImmediatePropagation} = \mathit{true} \\ \Sigma' := \Sigma[!\Sigma.\mathcal{N} := [], !\Sigma.\mathcal{H} := [], !\Sigma.\mathcal{P} := \mathcal{B}] \end{array}}{\Sigma, \theta \twoheadrightarrow \Sigma', \theta}$$

Figure 2: Semantics of event handling

capture phase, which is the reverse of the propagation path, $\mathcal{A}$.

`Capture-to-Target`: If the current phase is capture ($\mathcal{C}$) and it has ended, i.e., there is currently no executing handler ($C_m = \phi$), and the node list $\mathcal{N}$ and the event handler list $\mathcal{H}$ are both empty, then this rule shifts the machine to the target phase. In the target phase, only the target node (N) has to be processed, so the pending node list $\mathcal{N}$ is initialized to $[!\Sigma.N]$.

`Target-to-Bubble`: This is similar to `Capture-to-Target` but transitions from the target phase to the bubble phase. In the bubble phase, all nodes on the propagation path $\mathcal{A}$ must be processed, so the pending node list $\mathcal{N}$ is initialized to $\mathcal{A}$.

`Bubble-to-Default`: Once the bubble phase ends, we transfer to the default phase, but only if the event E's flag `defaultPrevented` is unset. In the default phase, default actions will be run on all nodes provided by the browser-specific function $getDefaults()$, so we initialize the pending node list $\mathcal{N}$ to the output of this function.[3]

`Default Prevented`: If at the end of the bubble phase E.`defaultPrevented` is set, then we skip the default phase. We set the phase to $\mathcal{D}$ and the pending node list and the pending handler list to empty. This emulates the end of the default phase.

`Remove Frame`: If there is no executing handler ($C_m = \phi$), the phase is set to $\mathcal{D}$ (default), and the pending node list $\mathcal{N}$ and the pending event handler list $\mathcal{H}$ are both empty, then the current event has been fully handled. So, the top (current) frame of the stack ($!\Sigma$) is removed.

`Get Event Handlers`: If no handler is currently executing ($C_m = \phi$), and the pending handler list $\mathcal{H}$ is empty, then all the handlers in the current node have been processed. So, we obtain the handlers for the next node in the pending node list $\mathcal{N}$ using the function `getEventHandlers`. This function takes as parameters the event, the (next) node, the phase and the heap and returns a list of event handlers. This rule applies only if the `stopPropagation` flag of the event is unset.

`Stop Propagation`: In the same starting state as the previous rule, if the `stopPropagation` flag of the event is set, then the current phase is set to $\mathcal{B}$ and the pending node list is set to empty. This new state emulates the end of the bubble phase and will immediately transition to the default phase by rule `Bubble-to-Default`, thus skipping any remaining parts of the capture, target and bubble phases.

`Run Event Handlers`: If there is no executing handler ($C_m = \phi$) and `stopImmediatePropagation` is unset, then this rule starts executing the next handler in the pending handler list $\mathcal{H}$. The notable aspect is that the $\mathcal{PC}$ of the new execution (the only frame in the new ρ) is obtained by joining the $\mathcal{PC}$s of the top frame of Σ ($!\Sigma.\ell$), the $\mathcal{PC}$ of the new handler ($\Gamma(f_m)$) and the $\mathcal{PC}$ when this handler was registered with this node (ℓ_c).

`Stop Immediate Propagation`: In the same initial state as the previous rule, if the `stopImmediatePropagation` flag of the event is set, then the current phase is set to $\mathcal{B}$ and the pending node list and the pending event handler list are both set to empty, directly bringing the state to the end of the bubble phase.

[3]In Safari, default actions are executed on the target node and its ancestors. The specification does not prescribe any specific list of nodes for the default phase.

B. Correctness of IFC

We formally prove soundness of our IFC instrumentation of our model of the DOM, the event loop and the sequential JS semantics by proving the standard property called termination insensitive noninterference. This property says that two runs with equal low-observable inputs produce equal low-observable outputs. We assume that the IFC enforcement on the sequential JS semantics satisfies some basic properties, which are described in an online appendix available from the authors' homepages. Roughly, the assumptions say that every small step of the sequential JS semantics must preserve a standard bisimulation relation between two runs of the program starting from low-equal memories. We describe these assumptions explicitly in the appendix and later discharge them for a specific instance of the sequential JS machine from [15] extended with the DOM API. The details of that extension are provided in our appendix. Here, we focus on the proofs of noninterference for our reactive model of event loops. For simplicity, we consider only a two-point security lattice $L < H$, but our definitions and theorems generalize to arbitrary lattices.

As usual, we define equivalence $\kappa_1 \sim \kappa_2$ of states $\kappa = \langle \Sigma, \theta \rangle$ of our transition relation.[4] To do this, we must also define the equivalence of sequential machine configurations C, of node and event handler lists and of stack frames. We show these definitions below. The definition of heap equivalence $\theta_1 \sim \theta_2$ is inherited from the sequential JS model and says that the parts of the heaps θ_1 and θ_2 reachable from their global objects by traversing only L-labelled pointers must be equal. Similarly, we also inherit definitions of call stack equivalence and $\mathcal{PC}$ stack equivalence, $\sigma_1 \sim \sigma_2$ and $\rho_1 \sim \rho_2$, from the sequential JS model.

Definition 1. *Two machines C_1 and C_2 are equivalent, written $C_1 \sim C_2$, if either one of the following hold:*

1) $C_1 = \langle \iota_1, \sigma_1, \rho_1 \rangle$, $C_2 = \langle \iota_2, \sigma_2, \rho_2 \rangle$ and
   a) $(\rho_1 \sim \rho_2)$
   b) $(\sigma_1 \sim \sigma_2)$

[4]Technically, $\sim$ is parametrized by a partial bijection β between names of heap locations allocated at corresponding points in the two runs. However, we omit the partial bijection here for readability. Our online appendix resolves the notation.

c) $(\Gamma(!\rho_1) = \Gamma(!\rho_2) = L \wedge \iota_1 = \iota_2)$ *or* $(\Gamma(!\rho_1) = \Gamma(!\rho_2) = H))$

2) $C_1 = C_2 = \phi$

Definition 2. *Two lists of nodes* $\mathcal{N}_1$ *and* $\mathcal{N}_2$ *are equivalent, written* $\mathcal{N}_1 \sim \mathcal{N}_2$, *iff* $|\mathcal{N}_1| = |\mathcal{N}_2| \wedge \forall i \in |\mathcal{N}_1|.\ \mathcal{N}_1[i] \sim \mathcal{N}_2[i]$.

Definition 3. *Two event handler lists* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *are equivalent, written* $\mathcal{H}_1 \sim \mathcal{H}_2$, *iff* $|\mathsf{low}(\mathcal{H}_1)| = |\mathsf{low}(\mathcal{H}_2)|$ *and for all* i, $\mathsf{low}(\mathcal{H}_1)[i] \sim \mathsf{low}(\mathcal{H}_2)[i]$ *where* low *is defined as:*

$$\mathsf{low}(nil) := nil$$

$$\mathsf{low}((\ell_c, eH) :: \mathcal{H}) := \begin{cases} eH :: \mathsf{low}(\mathcal{H}) & \text{if} \quad \ell_c = L \\ \mathsf{low}(\mathcal{H}) & \text{if} \quad \ell_c \neq L \end{cases}$$

The above definition states that two event handler lists are low-equal when the subsequences of low-visible event handlers in them are equal. (Note that event handlers eH are JS function objects, so the $\mathsf{low}(\mathcal{H}_1)[i] \sim \mathsf{low}(\mathcal{H}_2)[i]$ is the equivalence on JS objects from the sequential model.)

Next we define the equivalence of stack frames ν and the equivalence of stacks Σ. Intuitively, two frames are equivalent if they are labeled H or each of their respective elements are equivalent.

Definition 4. *Two frames* ν_1 *and* ν_2 *in the stack of machine configurations are equivalent, written* $\nu_1 \sim \nu_2$, *if either:*

1) $\Gamma(\nu_1) = H \wedge \Gamma(\nu_2) = H$ *or*
2) $\nu_1.C \sim \nu_2.C,\ \Gamma(\nu_1) = \Gamma(\nu_2),\ \nu_1.E \sim \nu_2.E,$ $\nu_1.N \sim \nu_2.N,\ \nu_1.\mathcal{A} \sim \nu_2.\mathcal{A},\ \nu_1.\mathcal{N} \sim \nu_2.\mathcal{N},$ $\nu_1.\mathcal{H} \sim \nu_2.\mathcal{H},\ \nu_1.\mathcal{P} = \nu_2.\mathcal{P}$ *and* *if* $\Gamma(!\rho(\nu_1.C)) = \Gamma(!\rho(\nu_2.C)) = L$, *then* $\nu_1.M = \nu_2.M$

Definition 5. *Given two stacks of machine configurations* Σ_1 *and* Σ_2, *suppose:*

1) ν_1 *is the first node in* Σ_1 *s.t.* $\Gamma(\nu_1) = H$
2) ν_2 *is the first node in* Σ_2 *s.t.* $\Gamma(\nu_2) = H$
3) Σ'_1 *is the prefix of* Σ_1 *up to but not including* ν_1
4) Σ'_2 *is the prefix of* Σ_2 *up to but not including* ν_2

Then, $\Sigma_1 \sim \Sigma_2$, *iff (1)* $|\Sigma'_1| = |\Sigma'_2|$, *and (2)* $\forall i \leq |\Sigma'_1|.\ (\Sigma'_1[i] \sim \Sigma'_2[i])$.

Definition 6 (κ-equivalence)**.** *Two states* $\kappa = \langle \Sigma, \theta \rangle$ *and* $\kappa' = \langle \Sigma', \theta' \rangle$ *are said to be equivalent, written* $\kappa \sim \kappa'$, *iff* $\Sigma \sim \Sigma'$ *and* $\theta \sim \theta'$.

We can now state our main noninterference theorem.

Theorem 1 (Termination-insensitive noninterference)**.** *If* $\kappa_1 \sim \kappa_2$, $\kappa_1 \twoheadrightarrow_\alpha \kappa'_1$ *and* $\kappa_2 \twoheadrightarrow_\alpha \kappa'_2$, *then either:*

- $\kappa'_1 \sim \kappa'_2$ *or*
- $\kappa'_1 \sim \kappa_2$ *or*
- $\kappa_1 \sim \kappa'_2$.

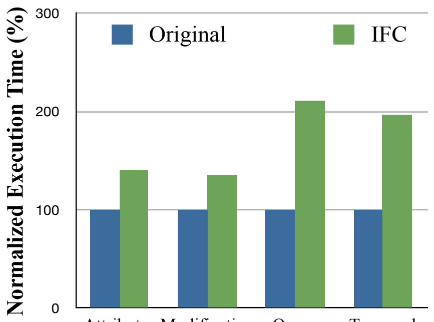

Figure 3: Overheads of IFC on Dromaeo DOM Core Benchmark Tests

We prove this theorem by setting a bisimulation relation $\kappa_1 \mathcal{R} \kappa_2$ which is equivalent to $\kappa_1 \sim \kappa_2$ and additionally provides enough structure to establish the theorem inductively. This relation is shown in our appendix, together with a proof of the theorem.

V. Implementation

We implement the IFC semantics described in Section IV in WebKit, a popular browser engine used in a number of browsers. Our implementation is built on top of the hybrid and optimized IFC implementation for JS bytecode from [15]. That implementation only instruments the sequential JS interpreter. We additionally instrument the DOM APIs, the event loop, and all native JS methods in the Array, RegExp, and String objects. Following [15], our implementation targets WebKit build #r122160 and works with the Safari web browser, version 6.0. All experiments are reported on a 3.2GHz Quad-core Intel Xeon processor with 8GB RAM, running Mac OS X version 10.7.4. Since [15] and our extension only target the bytecode interpreter, we disable JIT in all our experiments.

We attach security labels to every node in the DOM graph and all its properties, including pointer to other nodes. We then add appropriate IFC checks in the native C code implementing all DOM APIs up to Level 3. Additional instrumentation carries the labels from the native C code to the JS interpreter, where IFC checks are already provided by [15]. We make similar changes to the event loop, labeling every event and event handler. Our work adds approximately 2,300 lines of code above the 4,500 lines of initial instrumentation from [15]. Our implementation stops the execution of the program when it detects an IFC violation but for the purpose of measuring performance overheads reported here, we ignore violations.

We evaluate our instrumentation on the Dromaeo DOM Core benchmark [44], which measures the performance of various operations on the DOM. We find an average overhead of approximately 71% over the uninstrumented

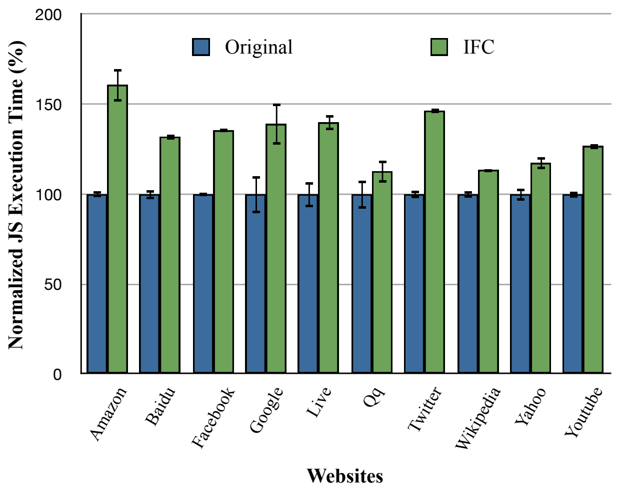

Figure 4: Overheads of IFC on Alexa Top 10 Websites

browser. Normalized overheads on different kinds of tests are shown in Figure 3 (standard deviations on individual tests were small, ranging from 0.17% to 8.45%). To get a more realistic evaluation, we also tested our instrumentation on the Alexa Top 10 websites [45]. We measured the execution time of the JS that loads initially on each website's front page, without any user interaction. The graph in Figure 4 shows normalized execution time. Error bars are standard deviations. The average overhead is approximately 32% and the worst overhead is around 60%. Note that both the Dromaeo and Alexa tests are very performance-intensive and do not count common browser delays like network communication and page rendering in the baseline. Compared to a baseline that includes these delays, our overheads are negligible.

Finally, as a sanity check, we run the very popular SunSpider benchmark [46]. SunSpider is a pure JS benchmark that does not cover events or the DOM, so it does not really execute the code paths we have instrumented. As a result, the overheads we get are similar to those of [15]. These overheads are included in our appendix.

VI. Related Work

We compare briefly to the most closely related work. Russo *et al.* [23] developed the first dynamic IFC monitor for an imperative language with DOM-like trees. In particular, they highlight information leaks via deletion and navigation of nodes in the DOM and show how their monitor can prevent them. The work does not cover live collections or event handling. Almeida-Matos *et al.* [24] provide an IFC monitor for a subset of the DOM (for a language similar to [23]) and also extend it for live collections. Their work covers only a part of the DOM and requires programmer provided annotations for handling live collections, which our method does not, as explained in Section III(c).

Hedin *et al.* [14], [20] develop a source-level interpreter for a core of JS with dynamic IFC checks. The interpreter is written from-scratch and provides an alternate code path for JS execution in a browser through a plug-in. In comparison, we build on [15], which works at the level of JS bytecode in WebKit and uses native WebKit code paths (which adds several order of magnitude less overhead). Besides, that work is limited to core JS without the DOM or events. COWL [19] is a recent system for improving security by confining JavaScript in web browsers. COWL uses coarse-grained labels at the level of browsing context (page, frame or workers) unlike our design that relies on fine-grained labels and, hence, offers higher precision. Kerschbaumer et al. [16] build an implementation of an information flow monitor for WebKit but do not handle all implicit flows. A black-box approach to enforcing non-interference is based on secure multi-execution (SME) [36]. Bielova *et al.* [17] and De Groef *et al.* [18] implement SME for web browsers. These systems do not attach labels to specific fields in the DOM. Instead, labels are attached to individual DOM APIs.

Gardner et al. [25] developed a formal specification for a minimal subset of DOM Level 1. Our formal specification goes beyond this and covers up to DOM Level 3. Lerner et al. [47] develop a formal model for the event handling mechanism of web browsers, but do not consider IFC. Our model additionally adds support for preemption of event handlers. The idea of developing a formal model for web browsers traces lineage to Featherweight Firefox [26], an OCaml model of the reactive core of a web browser. Both the DOM and the event handling mechanism are modeled, but abstractly, e.g., preemption is not modeled, nor are many DOM APIs.

VII. Conclusion

Although information flow control (IFC) for web browsers is a well-studied topic, two central browser aspects — event handling and the DOM — have not received enough attention. As we observe, both event handling and the DOM can be a source of subtle information flow leaks. Accordingly, we develop a formal model of event handling with preemption and of the DOM APIs up to Level 3, both fully instrumented for IFC. We prove our instrumentation sound, implement our ideas in WebKit and observe moderate overhead due to our IFC checks.

Acknowledgments

We thank the anonymous reviewers for their excellent feedback. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS^3), and the German Federal Ministry of Education and Research (BMBF) within the Centre for IT-Security, Privacy and Accountability (CISPA) at Saarland University.

References

[1] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proc. ACM Conference on Computer and Communications Security*, 2010, pp. 270–283.

[2] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Proc. IEEE Symposium on Security and Privacy*, 2012, pp. 413–427.

[3] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the Web," in *Proc. International World Wide Web Conference*, 2009, pp. 961–970.

[4] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proc. ACM Conference on Computer and Communications Security*, 2012, pp. 736–747.

[5] A. Barth, "The web origin concept." [Online]. Available: http://tools.ietf.org/html/rfc6454

[6] "Content Security Policy 1.0." [Online]. Available: http://www.w3.org/TR/CSP/

[7] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do – a large-scale study of the use of eval in JavaScript applications," in *Proc. European Conference on Object-Oriented Programming*, 2011, pp. 52–78.

[8] "Google Caja - A source-to-source translator for securing JavaScript-based web content." [Online]. Available: http://code.google.com/p/google-caja/

[9] "Facebook. FBJS." [Online]. Available: https://developers.facebook.com/docs/javascript

[10] D. Crockford, "ADSafe." [Online]. Available: http://adsafe.org/

[11] S. Maffeis and A. Taly, "Language-based isolation of untrusted javascript," in *Proc. IEEE Computer Security Foundations Symposium*, 2009, pp. 77–91.

[12] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proc. Annual Computer Security Applications Conference*, 2012, pp. 1–10.

[13] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information flow analysis for JavaScript," in *Proc. ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011, pp. 9–18.

[14] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *Proc. IEEE Computer Security Foundations Symposium*, 2012, pp. 3–18.

[15] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in WebKit's JavaScript bytecode," in *Proc. Principles of Security and Trust*, 2014, pp. 159–178.

[16] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz, "Towards precise and efficient information flow control in web browsers," in *Proc. Trust and Trustworthy Computing*, 2013, pp. 187–195.

[17] N. Bielova, D. Devriese, F. Massacci, and F. Piessens, "Reactive non-interference for a browser model," in *Proc. International Conference on Network and System Security*, 2011, pp. 97–104.

[18] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: a web browser with flexible and precise information flow control," in *Proc. ACM Conference on Computer and Communications Security*, 2012, pp. 748–759.

[19] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining JavaScript with COWL," in *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 131–146.

[20] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *Proc. ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[21] "Html 5 spec." [Online]. Available: http://www.w3.org/TR/html5

[22] "W3C Document Object Model Level 3 Core Specification." [Online]. Available: http://www.w3.org/TR/DOM-Level-3-Core

[23] A. Russo, A. Sabelfeld, and A. Chudnov, "Tracking information flow in dynamic tree structures," in *Proc. European Symposium on Research in Computer Security*, 2009, pp. 86–103.

[24] A. Almeida-Matos, J. Fragoso Santos, and T. Rezk, "An information flow monitor for a core of dom," in *Proc. Trustworthy Global Computing*, 2014, pp. 1–16.

[25] P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty, "DOM: Towards a formal specification," in *Proc. ACM SIGPLAN Workshop on Programming Language Technologies for XML*, 2008.

[26] A. Bohannon and B. C. Pierce, "Featherweight Firefox: Formalizing the core of a web browser," in *Proc. USENIX conference on Web Application Development*, 2010, pp. 11–22.

[27] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proc. European Symposium on Research in Computer Security*, 2008, pp. 333–348.

[28] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[29] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan 1996.

[30] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 79–90.

[31] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[32] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2009, pp. 113–124.

[33] ——, "Permissive dynamic information flow analysis," in *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2010, pp. 1–12.

[34] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proc. IEEE Computer Security Foundations Symposium*, 2009, pp. 43–59.

[35] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proc. Perspectives of Systems Informatics*, 2010, pp. 352–365.

[36] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. IEEE Symposium on Security and Privacy*, 2010, pp. 109–124.

[37] 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*, 2007.

[38] G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Proc. Asian Computing Science Conference on Secure Software*, 2006, pp. 75–89.

[39] G. Le Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *Proc. IEEE Computer Security Foundations Symposium*, 2007, pp. 218–232.

[40] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[41] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proc. IEEE Computer Security Foundations Symposium*, 2010, pp. 186 –199.

[42] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, August 2002.

[43] "W3C Document Object Model Level 3 Events Specification." [Online]. Available: http://www.w3.org/TR/DOM-Level-3-Events

[44] "Dromaeo: JS performance testing." [Online]. Available: http://dromaeo.com/

[45] "Alexa top 500 global sites." [Online]. Available: http://www.alexa.com/topsites

[46] "Sunspider 1.0.2 javascript benchmark." [Online]. Available: http://www.webkit.org/perf/sunspider/sunspider.html

[47] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q. de la Vallee, and S. Krishnamurthi, "Modeling and reasoning about dom events," in *Proc. USENIX Conference on Web Application Development*, 2012, pp. 1–12.