Modular Monitor Extensions for Information Flow Security in JavaScript

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

1 **Imperial College London**
`jose.fragoso.santos@imperial.ac.uk`
2 **Inria**
`tamara.rezk@inria.fr`
3 **University of Lisbon, SQIG-Instituto de Telecomunicações**
`ana.matos@ist.utl.pt`

Abstract

Client-side JavaScript programs often interact with the web page into which they are included, as well as with the browser itself, through APIs such as the DOM API, the XMLHttpRequest API, and the W3C Geolocation API. Precise reasoning about JavaScript security must therefore take API invocation into account. However, the continuous emergence of new APIs, and the heterogeneity of their forms and features, renders API behavior a moving target that is particularly hard to capture. To tackle this problem, we propose a methodology for modularly extending sound JavaScript information flow monitors with a generic API. Hence, to verify whether an extended monitor complies with the proposed noninterference property requires only to prove that the API satisfies a predefined set of conditions. In order to illustrate the practicality of our methodology, we show how an information flow monitor-inlining compiler can take into account the invocation of arbitrary APIs, without changing the code or the proofs of the original compiler. We provide an implementation of such a compiler with an extension for handling a fragment of the DOM Core Level 1 API. Furthermore, our implementation supports the addition of monitor extensions for new APIs at runtime.

1 Introduction

Isolation properties guarantee protection for trusted JavaScript code from malicious code. The noninterference property [9] provides the mathematical foundations for reasoning precisely about isolation. In particular, noninterference properties guarantee absence of flows from confidential/untrusted resources to public/trusted ones.

Although JavaScript can be used as a general-purpose programming language, many JavaScript programs are designed to be executed in a browser in the context of a web page. Such programs often interact with the web page in which they are included, as well as with the browser itself, through Application Programming Interfaces (APIs). Some APIs are fully implemented in JavaScript, whereas others are built with a mix of different technologies, which can be exploited to conceal sophisticated security violations. Thus, understanding the behavior of client-side web applications, as well as proving their compliance with a given security policy, requires cross-language reasoning. The size, complexity, and number of commonly used APIs poses an important challenge to any attempt at formally reasoning about the security of JavaScript programs [13]. To tackle this problem, we propose a methodology for extending JavaScript monitored semantics. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, we make it possible to prove that a security monitor is still noninterferent when extending it with a new API, without having to revisit the whole model. Generally, an API can be viewed as a particular set of specifications that a program can follow to make use of the

resources provided by another particular application. For client-side JavaScript programs, this definition of API applies both to: (1) interfaces of services that are provided to the program by the environment in which it executes, namely the web browser (for instance, the DOM, the XMLHttpRequest, and the W3C Geolocation APIs); (2) interfaces of JavaScript libraries that are explicitly included by the programmer (for instance, jQuery, Prototype.js, and Google Maps Image API). In the context of this work, the main difference between these two types of APIs is that in the former case their semantics escapes the JavaScript semantics, whereas in the latter it does not. The methodology proposed here was designed as a generic way of extending security monitors to deal with the first type of APIs. Nevertheless, we can also apply it to the second type whenever we want to execute the library's code in the original JavaScript semantics instead of the monitored semantics.

▶ **Example 1** (Running example: A Queue API)**.** Consider the following API for creating and manipulating priority queues. The API is available to the programmer through the global variable *queueAPI*, and variable *queueObj* is bound to a concrete queue:

queueAPI.queue(): creates a new priority queue;
queueObj.push(*el*, *priority*): adds a new element to the queue;
queueObj.pop(): pops the element with the highest priority.

The method calls from this API cannot be verified by the JavaScript monitor, as we are assuming that the code of the methods is not available to the JavaScript engine. Furthermore, the specification of the queue API may not obey the JavaScript semantics and hence prevention of the security leaks may need different constraints.

In order to extend a JavaScript security monitor to control the behavior of this API, one has to define what we call an *API Register* to set the security constraints associated to the corresponding API method calls on *queueAPI* and *queueObj*. API method calls should be implemented as interception points of the monitor semantics and the API Register should then make the invocation of these methods if the security constraints are satisfied.

The following questions then arise: What constraints must we impose on the new API register in order to preserve the noninterference guarantees of the JavaScript monitor? Is it possible to modularly prove noninterference of the extended monitor without revisiting the whole set of constraints, including those of the JavaScript monitor?

There are two main approaches for implementing a monitored JavaScript semantics: either one modifies a JavaScript engine so that it also implements the security monitor (as in [15]), or one inlines the monitor in the original program (as in [16], [8], and [10]). Both these approaches suffer from the problem of requiring ad-hoc security mechanisms for all targeted APIs. We show how to extend an information flow monitor-inlining compiler so that it also takes into account the invocation of APIs. Our extensible compiler requires each API to be associated with a set of JavaScript methods that we call its *IFlow Signature*, which describes how to handle the information flows triggered by its invocation. We provide a prototype of the compiler, which is available online [20]. A user can easily extend it by loading new IFlow signatures. Using the compiler, we give realistic examples of how to prevent common security violations that arise from the interaction between JavaScript and the DOM API. In a nutshell, the benefit of our approach is that it allows us to separate the proof of security for each API from the proof of security for the core language. This separation is, to the best of our knowledge, new and useful as new APIs are continuously emerging.

The contributions of the paper are: (1) a methodology for extending JavaScript monitors with API monitoring (Section 3.2), (2) the design of an extensible information flow monitor-inlining compiler that follows our methodology (Section 4), (3) an implementation [20] of a

JavaScript information flow monitor-inlining compiler (Section 5) that handles an important subset of the DOM API and is extensible with new APIs by means of IFlow Signatures.

2 Related Work

We refer the reader to a recent survey [7] on web scripts security and to [19] for a complete survey on information flow enforcement mechanisms up to 2003, while focusing here on the most closely related work on dynamic mechanisms for enforcing noninterference.

Flow-sensitive monitors for enforcing noninterference can be divided into *purely dynamic monitors* [3–5] and *hybrid monitors* [12, 22]. While hybrid monitors use static analysis to reason about untaken execution paths, purely dynamic monitors do not rely on any kind of static analysis. There are three main strategies in designing sound purely dynamic information flow monitors. The *no-sensitive-upgrade* (NSU) strategy [3] forbids the update of public resources inside private contexts. The *permissive-upgrade* strategy [4] allows sensitive upgrades, but forbids programs to branch depending on values upgraded in private contexts. Finally, the *multiple facet* strategy [5] makes use of values that appear differently to observers at different security levels. Here, we show how to extend information flow monitors that follow the NSU discipline.

Hedin and Sabelfeld [15] are the first to propose a runtime monitor for enforcing noninterference for JavaScript. The technique that we present for extending security monitors can be applied to this monitor, which is purely dynamic and follows the NSU discipline. In [14], the authors implement their monitor as an extended JavaScript interpreter. Their implementation makes use of the informal concepts of shallow and deep information flow models in order to cater for the invocation of built-in libraries and DOM API methods. However, these concepts are not formalised. In fact, our definition of monitored API can be seen as a formalisation of the notion of deep information flow model for libraries.

Both Chudnov and Naumann [8] and Magazinius et al. [16] propose the inlining of information flow monitors for simple imperative languages. In [10], we present a compiler that inlines a purely dynamic information flow monitor for a realistic subset of JavaScript. In the implementation presented in this paper we extend the inlining compiler of [10] with the DOM API, applying the methodology proposed here.

Taly et al. [21] study API confinement. They provide a static analysis designed to verify whether an API may leak its confidential resources. Unlike us, they only target APIs implemented in JavaScript, whose code is available for either runtime or static analysis.

Russo et al. [18] present an information flow monitor for a WHILE language with primitives for manipulating DOM-like trees and prove it sound. They do not model references. In [2], we present an information flow monitor for a simple language that models a core of the DOM API based on the work of Gardner et al. [11]. In contrast to [18], we can handle references and live collections. Here, we apply the techniques of [2] to develop monitor extensions for a fragment of the DOM Core Level 1 API [17]. Recent work [23] presents an information flow monitor for JavaScript extended with the DOM API that also considers event handling loops. To the best of our knowledge, no prior work proposes a generic methodology to extend JavaScript monitors and inlining compilers with arbitrary web APIs.

3 Modular Extensions for JavaScript Monitors

In this section we show how to extend a noninterferent monitor so that it takes into account the invocation of web APIs, while preserving the noninterference property.

3.1 Noninterferent JavaScript Monitors

JavaScript Memory Model. In JavaScript [1], objects can be seen as partial functions mapping strings to values. The strings in the domain of an object are called its properties. Memories are mappings from references to objects. In the following, we assume that memories include a reference to a special object called the *global object* pointed to by a fixed reference $\#glob$, that binds global variables. In this presentation, objects, properties, memories, references and values, are ranged over by o, p, μ, r and v, respectively.

We use the notation $[p_0 \mapsto v_0, \ldots, p_n \mapsto v_n]$ for the partial function that maps p_i to v_i where $i = 0, \ldots n$, and $f\,[p_0 \mapsto v_0, \ldots, p_n \mapsto v_n]$ for the function that coincides with f everywhere except in p_0, ..., p_n, which are otherwise mapped to v_0, ..., v_n respectively. Furthermore, we denote by $\mathsf{dom}(f)$ the domain of a function f, and by $f|_P$ the restriction of f to P (when $P \subseteq \mathsf{dom}(f)$). Finally, we write $f(r)(p)$ instead of $(f(r))(p)$, the application of the image of r by function f to p.

Sequences are denoted by stacking an arrow as in $\overrightarrow{v}$, and ϵ denotes the empty sequence. The length of $\overrightarrow{v}$ is given by $|\overrightarrow{v}|$ and $\cdot$ denotes concatenation of sequences.

Security Setting. Information flow policies such as noninterference are specified over security labelings that assign security levels, taken from a given security lattice, to the observable resources of a program. In the following, we use a fixed lattice $\mathcal{L}$ of security levels ranged over by σ. We denote by $\leq$ its order relation, by $\sigma_0 \sqcup \sigma_1$ the least upper bound between levels σ_0 and σ_1, and by $\sqcup\vec{\sigma}$ the least upper bound of all levels in the sequence $\vec{\sigma}$. In the examples, we consider two security levels $\{H, L\}$ such that $L < H$, meaning that resources labeled with *high* level H are more confidential than those labeled with *low* level L.

In our setting, a security labeling is as a pair $\langle \Gamma, \Sigma \rangle$, where Γ maps references, followed by properties, to security levels, and Σ maps references to security levels. Then, given an object o pointed to by a reference r, if defined, $\Gamma(r)(p)$ corresponds to the security levels associated with o's property p, and $\Sigma(r)$ with o's domain. The latter, also referred to as o's *structure security level*, controls the observability of the existence of properties [15].

We say that memory μ is well-labeled by $\langle \Gamma, \Sigma \rangle$ if $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Sigma) \subseteq \mathsf{dom}(\mu)$ and for every reference $r \in \mathsf{dom}(\Gamma)$, $\mathsf{dom}(\Gamma(r)) \subseteq \mathsf{dom}(\mu(r))$.

Security Monitor. JavaScript programs are statements, that include expressions, ranged over by s and e, respectively. We model an information flow monitor as a small-step semantics relation $\rightarrow_{\mathsf{IF}}$ between configurations of the form $\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle$ composed of (1) a memory μ (2) a statement s, that is to execute, (3) a sequence of security levels $\overrightarrow{\sigma_{\mathsf{pc}}}$, matching the expressions on which the original program branched to reach the current context, (4) a security labeling $\langle \Gamma, \Sigma \rangle$, and (5) a sequence of security levels $\overrightarrow{\sigma}$ matching the reading effects of the subexpressions of the expression being computed.

The *reading effect* [19] of an expression is defined as the least upper bound on the security levels of the resources on which the value to which it evaluates depends. Additionally, we assume that the reading effect of an expression is always higher than or equal to the level of the context in which it is evaluated, $\sqcup\overrightarrow{\sigma_{\mathsf{pc}}}$.

Low-equality. In order to account for a non-deterministic memory allocator, we rely on a partial injective function which relates observable references that point to the same resource in different executions of the same program [6]. The β relation is extended to relate observable values via the *β-equality*, which is denoted $\sim_\beta$: two objects are β-equal if they have the same domain and all their corresponding properties are β-equal; primitive values and parsed functions are β-equal if syntactically equal; and, two references r_0 and r_1 are β-equal if the latter is the image by β of the former.

Two memories μ_0 and μ_1 are said to be *low-equal* with respect to labelings $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, a security level σ, and a partial injective function β, written $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$, if μ_0 and μ_1 are well-labeled by $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$ respectively, and for all references $r_0, r_1 \in \mathsf{dom}(\beta)$, such that $r_1 = \beta(r_0)$, the following hold:

1. The observable domains (i.e. set of observable properties) of the objects pointed by r_0, r_1, coincide: $P_\sigma = \{p \in \mathsf{dom}(\mu_0(r_0)) \mid \Gamma_0(r_0)(p) \leq \sigma\} = \{p \in \mathsf{dom}(\mu_1(r_1)) \mid \Gamma_1(r_1)(p) \leq \sigma\}$;
2. The objects pointed by r_0, r_1 coincide in their observable domain: $\mu_0(r_0)|_P \sim_\beta \mu_1(r_1)|_P$;
3. If the structure security level of either object pointed by r_0, r_1 is observable ($\Sigma_0(r_0) \leq \sigma$ or $\Sigma_1(r_1) \leq \sigma$), then their domains and structure security levels coincide: $\mathsf{dom}(\mu_0(r_0)) = \mathsf{dom}(\mu_1(r_1))$ and $\Sigma_0(r_0) = \Sigma_1(r_1)$.

We extend informally the definition of low-equality to sequences of labeled values and to program continuations (the interested reader can find the formal definitions in [20]). Two sequences of labeled values are low-equal with respect to a given security level σ, denoted $\vec{v_0}, \vec{\sigma_0} \approx_{\beta,\sigma} \vec{v_1}, \vec{\sigma_1}$ if for each position of both sequences, either the two values in that position are low-equal, or the levels that are associated with both of them are not observable. Low-equality between program continuations $s_0, \overrightarrow{\sigma_{pc0}}, \vec{\sigma_0} \approx_{\beta,\sigma} s_1, \overrightarrow{\sigma_{pc1}}, \vec{\sigma_1}$ relaxes syntactic equality between programs in order to relate the intermediate states of the execution of the same original program in two low-equal memories, as illustrated by the following example.

▶ **Example 2** (Low-equal program continuations)**.** Consider the program $x = y$, an initial labeling $\langle \Gamma, \Sigma \rangle$ such that $\Gamma(\#glob)(x) = \Gamma(\#glob)(y) = H$, and two memories μ_0 and μ_1 such that $\mu_i = [\#glob \mapsto [x \mapsto \mathit{undefined}, y \mapsto i]]$, for $i \in \{0, 1\}$. The execution of one computation step of this program in μ_0 and μ_1 yields the programs $x = 0$ and $x = 1$. Since the reading effects associated with the values 0 and 1 are both H, the expressions $x = 0$ and $x = 1$ are low-equal. Formally: $x = 0, \langle L \rangle, \langle H \rangle \approx_{\mathsf{id},L} x = 1, \langle L \rangle, \langle H \rangle$ (where id is the identity function).

Finally, two monitor configurations $\langle \mu_0, s_0, \overrightarrow{\sigma_{pc0}}, \Gamma_0, \Sigma_0, \vec{\sigma_0} \rangle$ and $\langle \mu_1, s_1, \overrightarrow{\sigma_{pc1}}, \Gamma_1, \Sigma_1, \vec{\sigma_1} \rangle$ are said to be *low-equal* w.r.t a level σ and function β, written $\langle \mu_0, s_0, \overrightarrow{\sigma_{pc0}}, \Gamma_0, \Sigma_0, \vec{\sigma_0} \rangle \approx_{\beta,\sigma} \langle \mu_1, s_1, \overrightarrow{\sigma_{pc1}}, \Gamma_1, \Sigma_1, \vec{\sigma_1} \rangle$, if $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \sigma_1$ and $s_0, \overrightarrow{\sigma_{pc0}}, \vec{\sigma_0} \approx_{\beta,\sigma} s_1, \overrightarrow{\sigma_{pc1}}, \vec{\sigma_1}$.

Noninterferent Monitor.

In the remaining of the paper, we consider only *noninterferent* JavaScript monitors. As usual, a monitor $\rightarrow_{\mathsf{IF}}$ is noninterferent, written $\mathbf{NI}_{\mathbf{mon}}(\rightarrow_{\mathsf{IF}})$, if its application on two low-equal configurations produces two low-equal configurations.

▶ **Definition 3** (Monitor Noninterference)**.** A monitor $\rightarrow_{\mathsf{IF}}$ is said to be *noninterferent*, written $\mathbf{NI}_{\mathbf{mon}}(\rightarrow_{\mathsf{IF}})$, if for every programs s_0, s_1, memories μ_0, μ_1, and labeling $\langle \Gamma, \Sigma \rangle$, such that μ_0, μ_1 are well-labeled by $\langle \Gamma, \Sigma \rangle$ and, for all security levels σ, there exists β such that $\langle \mu_0, s_0, \epsilon, \Gamma, \Sigma, \epsilon \rangle \approx_{\beta,\sigma} \langle \mu_1, s_1, \epsilon, \Gamma, \Sigma, \epsilon \rangle$, if $\langle \mu_0, s_0, \epsilon, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{\mathsf{IF}} \langle \mu_0', v_0', \epsilon, \Gamma', \Sigma', \sigma' \rangle$ and $\langle \mu_1, s_1, \epsilon, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{\mathsf{IF}} \langle \mu_1', v_1', \epsilon, \Gamma', \Sigma', \sigma' \rangle$ then there is an extension β' of β such that $\langle \mu_0', v_0', \epsilon, \Gamma', \Sigma', \sigma' \rangle \approx_{\beta,\sigma} \langle \mu_1', v_1', \epsilon, \Gamma', \Sigma', \sigma' \rangle$.

3.2 API Extensions to JavaScript Monitors

API relation. Even if the execution of certain APIs escapes the JavaScript semantics, the interaction between JavaScript programs and these APIs is mediated via special API objects that exist in the JavaScript memory. In the following, we assume that (1) the state of the API can be fully encoded in a JavaScript memory and (2) the behavior of each API method only depends on its state. An API is thus modeled as a semantic relation $\Downarrow^{\mathsf{JS}}_{\mathsf{API}}$ of the form $\langle \mu, \overrightarrow{v} \rangle \Downarrow^{\mathsf{JS}}_{\mathsf{API}} \langle \mu', v' \rangle$ where μ is the JavaScript memory in which the API is executed, μ'

is the resulting memory, the sequence of values $\overrightarrow{v}$ corresponds to the arguments of the API invocation, and v' is the value to which the API invocation evaluates. Accordingly, a *monitored API relation*, $\Downarrow_{\mathsf{API}}$, has the form

$$\langle \mu, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \ \Downarrow_{\mathsf{API}} \ \langle \mu', \Gamma', \Sigma', v, \sigma \rangle$$

which adds to the original API configuration the initial and final labelings $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$ (respectively), the sequence of security levels $\overrightarrow{\sigma}$ that is associated with the arguments of the API invocation, and their corresponding reading effect σ.

API register. The bridge between API invocations and the corresponding monitored API semantics is performed by a *API register*, denoted by $\mathcal{R}_{\mathsf{API}}$. We define an API register as a function that, given a memory and a sequence of values, returns a monitored API relation.

▶ **Example 4** (Queue API Register)**.** In order for an extended monitor to take into account the methods of the Queue API from Example 1, the API Register must be extended to handle invocations of the Queue API methods. In the following, $\Downarrow_{QU}$, $\Downarrow_{PU}$, and $\Downarrow_{PO}$ are the API relations corresponding to each one of the methods of the Queue API:

$$\mathcal{R}_Q(\mu, \langle r, m, \ldots \rangle) = \begin{cases} \Downarrow_{QU} & \text{if } m = \text{“queue”} \wedge \$q \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{PU} & \text{if } m = \text{“push”} \wedge \$q \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{PO} & \text{if } m = \text{“pop”} \wedge \$q \in \mathsf{dom}(\mu(r)) \end{cases}$$

The idea is to “mark” the Queue API object (the one bound to variable *queueAPI*) as well as the concrete queue objects, with a special property (in this case, $\$q$).

Monitor-extending Constructor. We now define a monitor-extending constructor $\mathcal{E}$ that, given a monitored small-step semantics $\rightarrow_{\mathsf{IF}}$, a partial function $\mathsf{Intercept}$ mapping statements to sequences of values, and an API register $\mathcal{R}_{\mathsf{API}}$, produces a new monitored small-step semantics $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$. The new extended semantics handles the invocation of APIs by applying the API relation that is returned by $\mathcal{R}_{\mathsf{API}}$. API invocation is triggered by *interception points*, statements containing expression redexes (expressions that only have values as subexpressions) and that are in the set $\mathsf{Intercept}$. Then, if the sequence of values to which its subexpressions evaluate is in the domain of the API register $\mathcal{R}_{\mathsf{API}}$, their image by $\mathcal{R}_{\mathsf{API}}$ is the semantic relation that models the API to be executed.

The definition of $\mathcal{E}$, given in Figure 1, makes use of a syntactic function, $\mathsf{SubExpressions}$, defined on JavaScript statements, such that $\mathsf{SubExpressions}[[s]]$ corresponds to the sequence comprising all the subexpressions of s in the order by which they are evaluated. Rules [Non-Intercepted Program Construct] and [Intercepted Program Construct - Standard Execution] model the case in which the new small-step semantics behaves according to the original semantics $\rightarrow_{\mathsf{IF}}$. Rule [Intercepted Program Construct - API Execution] models the case in which an API is executed. The semantics rule retrieves the semantics relation that models the API to execute (using the API register) and then executes the API. After executing the API, the sequence of values of its subexpressions is replaced with the value to which the API call evaluates. Analogously, the sequence of levels of its subexpressions is replaced with the reading effect of the API call.

3.3 Sufficient Conditions for Noninterferent API Extensions

We identify sufficient conditions to be satisfied by API relations in order for the new extended monitored semantics $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$ to be noninterferent, assuming that the original monitor $\rightarrow_{\mathsf{IF}}$ is noninterferent.

$$\textsc{Non-Intercepted Program Construct}\quad \frac{s \notin \mathsf{Intercept} \qquad \langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \rightarrow_{\mathsf{IF}} \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}$$

$$\textsc{Intercepted Program Construct - Standard Execution}\quad \frac{s \in \mathsf{Intercept} \qquad (\mu, \mathsf{SubExpressions}[[s]]) \notin \mathsf{dom}(\mathcal{R}_{\mathsf{API}}) \qquad \langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \rightarrow_{\mathsf{IF}} \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma_{\mathsf{pc}}}' \rangle}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}$$

$$\textsc{Intercepted Program Construct - API Execution}\quad \frac{\begin{array}{c} s \in \mathsf{Intercept} \qquad (\mu, \mathsf{SubExpressions}[[s]]) \in \mathsf{dom}(\mathcal{R}_{\mathsf{API}}) \\ |\mathsf{SubExpressions}[[s]]| = n+1 \qquad \overrightarrow{\sigma} = \overrightarrow{\sigma}' \cdot \langle \sigma_0, \ldots, \sigma_n \rangle \qquad \Downarrow_{\mathsf{API}} = \mathcal{R}_{\mathsf{API}}(\mu, \mathsf{SubExpressions}[[s]]) \\ \langle \mu, \Gamma, \Sigma, \mathsf{SubExpressions}[[s]], \langle \sigma_0, \ldots, \sigma_n \rangle \rangle\ \Downarrow_{\mathsf{API}}\ \langle \mu', \Gamma', \Sigma', v', \sigma' \rangle \end{array}}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', v', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \cdot \sigma' \rangle}$$

Figure 1 Definition of the monitor-extending constructor $\mathcal{E}$.

The first condition requires that the API relation is *confined*, as formalized in Definition 5. An API relation is *confined* if it only creates/updates resources whose levels are higher than or equal to the least upper bound on the levels of its arguments. This constraint is needed because the choice of which API to execute may depend on all of its arguments.

▶ **Definition 5** (Confined API Relation/Register)**.** An API relation $\Downarrow_{\mathsf{API}}$ is confined if, for every memory μ well-labeled by a labeling $\langle \Gamma, \Sigma \rangle$, every sequence of argument values $\overrightarrow{v}$ and corresponding sequence of security levels $\overrightarrow{\sigma}$, if $\langle \mu, \langle \Gamma, \Sigma \rangle, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \Downarrow_{\mathsf{API}} \langle \mu', \langle \Gamma', \Sigma' \rangle, v', \sigma' \rangle$ for some memory μ', labeling $\langle \Gamma', \Sigma' \rangle$, value v', and level σ'; then, for all security levels $\hat{\sigma}$:

$$\sqcup \overrightarrow{\sigma} \not\leq \hat{\sigma} \quad \Rightarrow \quad \mu, \Gamma, \Sigma \approx_{\mathsf{id}, \hat{\sigma}} \mu', \Gamma', \Sigma' \ \wedge\ \sigma' \not\leq \hat{\sigma}$$

Furthermore, we say that the API Register function $\mathcal{R}_{\mathsf{API}}$ is confined, written $\mathbf{Conf}(\mathcal{R}_{\mathsf{API}})$, if all the API relations in its range are confined, and if for every memories μ and μ', labelings $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$, sequence of values $\overrightarrow{v}$, security level σ, and function β, such that $\mu, \Gamma, \Sigma \approx_{\beta, \sigma} \mu', \Gamma', \Sigma'$, then $\mathcal{R}_{\mathsf{API}}(\mu, \overrightarrow{v}) = \mathcal{R}_{\mathsf{API}}(\mu', \beta(\overrightarrow{v}))$.

The second condition requires that the API relation is *noninterferent*, as formalized in Definition 6. In order to relate the outputs of the API Register in two low-equal memories, we extend the notion of low-equality to API registers. Informally, two API registers are said to be low-equal if, whenever they are given as input two low-equal memories and two low-equal sequences of values, they output the same noninterferent API relation. Then, an API relation is *noninterferent* if whenever it is executed on two low-equal memories, it produces two low-equal memories and two low-equal values.

▶ **Definition 6** (Noninterferent API Relation/Register)**.** An API relation $\Downarrow_{\mathsf{API}}$ is said to be noninterferent, written $\mathbf{NI}(\Downarrow_{\mathsf{API}})$, if for every two memories μ_0 and μ_1 respectively well-labeled by $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, any two sequences of values $\overrightarrow{v_0}$ and $\overrightarrow{v_1}$, respectively labeled by two sequences of security levels $\overrightarrow{\sigma_0}$ and $\overrightarrow{\sigma_1}$, and any security level σ for which there exists a function β such that $\overrightarrow{v_0}, \overrightarrow{\sigma_0} \approx_{\beta, \sigma} \overrightarrow{v_1}, \overrightarrow{\sigma_1}$ and $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta, \sigma} \mu_1, \Gamma_1, \Sigma_1$, if:

$$\langle \mu_0, \Gamma_0, \Sigma_0, \overrightarrow{v_0}, \overrightarrow{\sigma_0} \rangle \Downarrow_{\mathsf{API}} \langle \mu_0', \Gamma_0', \Sigma_0', v_0', \sigma_0' \rangle \ \wedge\ \langle \mu_1, \Gamma_1, \Sigma_1, \overrightarrow{v_1}, \overrightarrow{\sigma_1} \rangle \Downarrow_{\mathsf{API}} \langle \mu_1', \Gamma_1', \Sigma_1', v_1', \sigma_1' \rangle$$

then there is an extension β' of β s.t. $\mu_0', \Gamma_0', \Sigma_0' \approx_{\beta', \sigma} \mu_1', \Gamma_1', \Sigma_1'$ and $\langle v_0' \rangle, \langle \sigma_0' \rangle \approx_{\beta', \sigma} \langle v_1' \rangle, \langle \sigma_1' \rangle$.

Furthermore, we say that the API Register function $\mathcal{R}_{\mathsf{API}}$ is noninterferent, written $\mathbf{NI}(\mathcal{R}_{\mathsf{API}})$, if all the API relations in its range are noninterferent.

▶ **Example 7** (Noninterferent JavaScript program using the Queue API)**.** Assume that the APIs given in Example 1 are noninterferent and consider the following program that starts by computing two objects `o0` and `o1`:

```
1  q = queueAPI.createQueue();
2  if (h) { q.push(o1, 1); }
3  q.push(o0, 0); l = q.pop();
```

If this program starts with memories μ_i ($i \in \{0,1\}$) using labeling $\langle \Gamma, \Sigma \rangle$ and assuming that in both executions the invocations of all the external APIs go through (i.e. the execution is never blocked), then it must terminate with memories μ_i' labeled by Γ', Σ:

$$\mu_i = \left[\begin{array}{c}(\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1,\\ (\#glob, h) \mapsto i\end{array}\right] \qquad \Gamma = \left[\begin{array}{c}(\#glob, h) \mapsto H, (\#glob, l) \mapsto L,\\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L\end{array}\right]$$

$$\mu_i' = \left[\begin{array}{c}(\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1,\\ (\#glob, h) \mapsto i, (\#glob, l) \mapsto r_i, (\#glob, q) \mapsto r_q\end{array}\right] \quad \Gamma' = \left[\begin{array}{c}(\#glob, h) \mapsto H, (\#glob, l) \mapsto H,\\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L\end{array}\right]$$

Since initial memories are low-equal, $\mu_0, \Gamma, \Sigma \approx_{\mathsf{id}, L} \mu_1, \Gamma, \Sigma$, we use the hypothesis that all three API relations are noninterferent to conclude that the memories yielded by the invocation of the API relations in lines `1`, `2`, and `3` are also low-equal. Furthermore, in the execution that maps `h` to 1, the value of `l` clearly depends on the value of `h`, from which we conclude that it is also the case in the execution that maps `h` to 0.

Our main result states that if the API relation is confined and noninterferent, then the extension of the noninterferent JavaScript monitor with the API monitor is noninterferent.

▶ **Theorem 8** (Security)**.** *For every monitored semantics* $\rightarrow_{\mathsf{IF}}$*, API register* $\mathcal{R}_{\mathsf{API}}$ *and set of interception points* Intercept*:*

$$\mathbf{NI_{mon}}(\rightarrow_{\mathsf{IF}}) \wedge \mathbf{NI}(\mathcal{R}_{\mathsf{API}}) \wedge \mathbf{Conf}(\mathcal{R}_{\mathsf{API}}) \Rightarrow \mathbf{NI_{mon}}(\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}}))$$

4 A Meta-Compiler for Securing Web APIs

We now propose a way of extending an information flow monitor inlining compiler to take into account the execution of arbitrary APIs.

Input compilers. We assume available two inlining compilers specified by compilation functions $\mathcal{C}_{\mathsf{e}}$ and $\mathcal{C}_{\mathsf{s}}$ for compiling JavaScript expressions and statements, respectively. Function $\mathcal{C}_{\mathsf{s}}$ makes use of function $\mathcal{C}_{\mathsf{e}}$. The compilers $\mathcal{C}_{\mathsf{e}}/\mathcal{C}_{\mathsf{s}}$ map every expression e/statement s to a pair $\langle s', i \rangle$, where:

1. statement s' simulates the execution of e/s in the monitored semantics;
2. index i is such that, after the execution of s', (1) the compiler variable $\$\hat{v}_i$ stores the value to which e/s evaluates in the original semantics and (2) the compiler variable $\$\hat{l}_i$ stores its corresponding reading effect.

We assume that the inlining compiler works by pairing up each variable/property with a new one, called its *shadow* variable/property [8, 16], that holds its corresponding security level. Since the compiled program has to handle security levels, we include them in the set of program values, which means adding them to the syntax of the language as such, as well as adding two new binary operators corresponding to $\leq$ (the order relation) and $\sqcup$ (the least upper bound). Besides adding to every object o an additional shadow property $\$l_p$ for every property p in its domain, the inlined monitoring code is also assumed to extend o with a special property $\$struct$ that stores its structure security level.

▶ **Example 9** (Instrumented Labeling)**.** Given an object $o = [p \mapsto v_0, q \mapsto v_1]$ pointed to by r_o and a labeling $\langle \Gamma, \Sigma \rangle$, such that $\Gamma(r_o) = [p \mapsto H, q \mapsto L]$ and $\Sigma(r_o) = L$, the instrumented counterpart of o labeled by $\langle \Gamma, \Sigma \rangle$ is $\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$struct \mapsto L]$.

$$\textsc{Intercepted Expression}$$

$$\frac{\begin{array}{c}\mathsf{SubExpressions}[[e]] = \langle e_0, \ldots, e_n\rangle \qquad \mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e_0\rangle = \langle s_0, i_0\rangle \ \cdots \ \mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e_n\rangle = \langle s_n, i_n\rangle \\ \hat{e} = \mathsf{Replace}[[e, \$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}]] \qquad \langle\mathcal{C}_{\mathsf{e}}\rangle\hat{e} = \langle\hat{s}, i\rangle \\ s_{\mathsf{api}} = \left\{\begin{array}{l} s_0 \ \ldots \ s_n \\ \$if_{\mathsf{sig}} = \$apiRegister(\$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}); \\ \mathsf{if}(\$if_{\mathsf{sig}})\{ \\ \quad \$if_{\mathsf{sig}}.check(\$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}, \$\hat{l}_{i_0}, \ldots, \$\hat{l}_{i_n}); \\ \quad \$\hat{v}_i = \hat{e}; \\ \quad \$\hat{l}_i = \$if_{\mathsf{sig}}.label(\$\hat{v}_i, \$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}, \$\hat{l}_{i_0}, \ldots, \$\hat{l}_{i_n}); \\ \} \ \mathsf{else} \ \{\hat{s}\} \end{array}\right. \qquad s' = \left\{\begin{array}{ll} s_{\mathsf{api}} & \text{if } \hat{e} \in \mathsf{Intercept} \\ \hat{s} & \text{otherwise} \end{array}\right. \end{array}}{\mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e\rangle = \langle s', i\rangle}$$

Figure 2 Extended Compiler $\mathcal{C}_{\mathsf{API}}$.

4.1 IFlow Signatures

We propose *IFlow signatures* to simulate monitored executions of API relations. IFlow signatures are composed of three methods – *domain*, *check*, and *label*. Method *domain* checks whether or not to apply the API, *check* checks if the constraints associated with the API are verified, and *label* updates the instrumented labeling and outputs the reading effect associated with a call to the API. Functions *check* and *label* must be specified separately because *check* has to be executed before calling the API (in order to prevent its execution when it can potentially trigger a security violation), whereas *label* must be executed after calling the API (so that it can label the memory resulting from its execution). Formally, we define an *IFlow Signature* as a triple $\langle\#check, \#label, \#domain\rangle$, where: $\#check$ is the reference of the *check* function object, $\#label$ is the reference of the *label* function object, and $\#domain$ is the reference of the *domain* function object.

Runtime API Register. We assume the existence of a runtime function called the *runtime API register*, that simulates the API Register, which we denote by $\$apiRegister$. The function $\$apiRegister$ makes use of the *domain* method of each API in its range to decide whether there is an API relation associated with its inputs, in which case it outputs an object containing the corresponding IFlow Signature, otherwise it returns *null*.

Meta-compiler. Figure 2 presents a new meta-compiler, $\mathcal{C}_{\mathsf{API}}$, that receives as input an inlining compiler for JavaScript expressions, $\mathcal{C}_{\mathsf{e}}$, and outputs a new inlining compiler that can handle the invocation of the APIs whose signatures are in the range of the API register simulated by $\$apiRegister$. Since statement redexes are not intercepted, the compilation function $\mathcal{C}_{\mathsf{s}}$ is left unchanged except that it uses the new compilation function for expressions for compiling the subexpressions of the given statement. The specification of the meta-compiler makes use of a syntactic function Replace that receives as input an expression and a sequence of variables and outputs the result of substituting each one of its subexpressions by the corresponding sequence variable. Intercept is the set of all statements that contain an expression redex whose execution is to be intercepted by the monitored semantics. Each expression that can be an interception point of the semantics is compiled by the compiler generated by the meta-compiler to a statement, which: (1) executes the statements corresponding to the compilation of its subexpressions, (2) tests if the sequence of values corresponding to the subexpressions of the expression to compile is associated with an IFlow signature, (3) if the test is true, it executes the *check* method of the corresponding IFlow signature, an expression equivalent to the original expression, and the *label* method of the corresponding IFlow signature. If the test is false, it executes the compilation of an

expression equivalent to the original one by the original inlining compiler. For simplicity, we do not take into account expressions that manipulate control flow, meaning that the evaluation of a given expression implies the evaluation of all its subexpressions. Therefore, we do not consider the JavaScript conditional expression. This limitation can be surpassed by re-writing all conditional expressions as IF statements before applying the compiler.

4.2 Correctness

We say that an inlining compiler is *correct* with respect to a given monitored semantics $\rightarrow_{\mathsf{IF}}$ if, provided that a program and its compiled counterpart are evaluated in "similar" memories, the evaluation of the original one in the monitored semantics terminates *if and only if* the evaluation of its compilation also terminates in the original semantics, in which case the final memories as well as the computed values are again "similar". Here we use a notion of *similarity* between labeled memories in the monitored semantics and instrumented memories in the original semantics, denoted by $\mathcal{S}_\beta$. This relation requires that for every object in the labeled memory, the corresponding labeling coincides with the instrumented labeling and that the property values of the original object be similar to those of its instrumented counterpart. (The formal definition of $\mathcal{S}_\beta$ can be found in the companion report [20].)

The correctness of the compiler generated by the meta-compiler depends on the correctness of the compiler given as input and the correctness of the IFlow signatures in the runtime API register. Definitions 10 and 11 formally specify the conditions that the instrumented API register must verify in order for the generated compiler to be correct. We use $\rightarrow^*_{\mathsf{JS}}$ as the semantics relation for JavaScript configurations.

▶ **Definition 10** (Correct IFlow Signature)**.** An IFlow Signature $\langle \#c, \#l, \#d \rangle$ is *correct* with respect to an API $\Downarrow_{\mathsf{API}}$ if for all memories μ_0 and μ_1, labeling $\langle \Gamma, \Sigma \rangle$, sequence of values $\overrightarrow{v}$, and sequence of security levels $\overrightarrow{\sigma}$, such that $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu_1$ for some function β, then: $\langle \mu_0, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \ \Downarrow_{\mathsf{API}} \ \langle \mu_0', \Gamma', \Sigma', v_0, \sigma \rangle$ if and only if (1) $\langle \mu_1, \#c(\beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1', true \rangle$, (2) $\langle \mu_1', \beta(\overrightarrow{v}) \rangle \ \Downarrow^{\mathsf{JS}}_{\mathsf{API}} \ \langle \mu_1'', v_1 \rangle$, and (3) $\langle \mu_1'', \#l(v_1, \beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1''', \sigma \rangle$, in which case $\langle \mu_0', \Gamma', \Sigma' \rangle \ \mathcal{S}_{\beta'} \ \mu_1'''$ and $v_0 \ \mathcal{S}_\beta \ v_1$, for some β' extending β.

▶ **Definition 11** (Correct Runtime API Register)**.** A runtime API register corresponding to a function object pointed by $\#\$apiRegister$ is correct with respect to an API register $\mathcal{R}_{\mathsf{API}}$ if for all memories μ_0 and μ_1, labeling $\langle \Gamma, \Sigma \rangle$ and sequence of values $\overrightarrow{v}$, such that $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu_1$ for some function β, then: $\mathcal{R}_{\mathsf{API}}(\mu_0, \overrightarrow{v}) = \Downarrow_{\mathsf{API}}$ if and only if (1) $\langle \mu_1, \#apiRegister(\beta(\overrightarrow{v})) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1', r_{\mathsf{sig}} \rangle$, (2) $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_{\beta'} \ \mu_1'$ for some β' extending β, and (3) signature $\langle o_{\mathsf{sig}}(\text{"check"}), o_{\mathsf{sig}}(\text{"label"}), o_{\mathsf{sig}}(\text{"domain"}) \rangle$ is correct with respect to $\Downarrow_{\mathsf{API}}$, where $o_{\mathsf{sig}} = \mu_1'(r_{\mathsf{sig}})$.

Theorem 12 states that provided that the compiler given as input to the meta-compiler is correct and the runtime API register is correct, the generated compiler is also correct.

▶ **Theorem 12** (Correctness)**.** *If compiler $\mathcal{C}$ is correct w.r.t. $\rightarrow_{\mathsf{IF}}$, then $\mathcal{C}_{\mathsf{API}}\langle \mathcal{C} \rangle$ is correct w.r.t. $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$ provided that the runtime API register is correct w.r.t. $\mathcal{R}_{\mathsf{API}}$.*

The meta-compiler proposed in this section allows the developer of the inlining compiler to extend it in a modular way, developing and proving each API IFlow signature at a time.

5 Implementation of the Meta Compiler and DOM API Extension

An implementation of a meta-compiler based on the JavaScript inlining compiler of [10] can be found in [20] together with an online testing tool and a set of IFlow signatures that

$$\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}(\mu, \langle r, m, \ldots\rangle) = \begin{cases} \Downarrow_{\mathsf{cre}} & \text{if } m = \text{“createElement”} \wedge r = \#doc \\ \Downarrow_{\mathsf{app}} & \text{if } m = \text{“appendChild”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{rem}} & \text{if } m = \text{“removeChild”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{len}} & \text{if } m = \text{“length”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{par}} & \text{if } m = \text{“parentNode”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{ind}} & \text{if } m \in \mathsf{Number} \wedge @\mathsf{tag} \in \mathsf{dom}(\mu(\mathsf{r})) \\ \Downarrow_{\mathsf{sib}} & \text{if } m = \text{“nextSibling”} \wedge @tag \in \mathsf{dom}(\mu(r)) \end{cases}$$

Figure 3 API register $\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}$ for the DOM API.

includes all those studied in the paper. As a case study, we give a high-level description of our the DOM API extension.

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [17]. In order to access the functionalities of this API, JavaScript programs manipulate a special kind of objects, here named *DOM objects*. In contrast to the ECMA Standard [1] that specifies in full detail the internals of objects created at runtime, the DOM API only specifies the behavior that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine. Instead, there is a separate engine, often called the *render engine*, whose role is to do so. Therefore, interactions between a JavaScript program and the DOM may potentially stop the execution of the JavaScript engine and trigger a call to the render engine. Thus, a monitored JavaScript engine has no access to the implementation of the DOM API.

We model DOM objects as standard JavaScript objects and we assume that every memory contains a *document* object denoted *doc*, which is accessed through the property “doc” and stored in fixed reference $\#doc$. Each DOM object defines a property $@tag$ that specifies its tag (for instance, $\langle div\rangle$, $\langle html\rangle$, $\langle a\rangle$) and, possibly, an arbitrary number of indexes 0, ..., n each pointing to one of its $n + 1$ children. DOM Element objects form a *forest*, such that the displayed HTML document corresponds to the tree hanging from the object pointed to by $\#doc$. Due to lack of space, we only present the labeled API relation for removing a DOM Element object from its parent object in the DOM forest. This API method gives rise to implicit information flows [2, 18, 23] that its labeled version needs to take into account.

▶ **Example 13** (Leak via removeChild - Order Leak)**.** Suppose that in the original memory there are three orphan DIV nodes bound to variables `div1`, `div2`, and `div3`.

```
1 div1.appendChild(div2); div1.appendChild(div3);
2 if(h) { div1.removeChild(div2); }
3 l = div1[0];
```

After the execution of this program, depending on the value of the high variable `h`, the value of the low variable `l` can be either that of `div2` or `div3`, meaning that the final level associated with variable `l` must be H in both executions. This example shows that, when removing a node, the new indexes of its right siblings are affected. To tackle this problem, the labelled DOM API methods enforce that the level of the property through which a DOM object is accessed is always lower than or equal to the levels of the properties corresponding to its right siblings.

Below we give the specification of the labeled API relation $\Downarrow_{\mathsf{rem}}$ for removing a DOM object from its parent in the DOM forest. This rule receives a sequence of arguments $\langle r_0, m_1, r_2\rangle$ as input and removes the object pointed to by r_2 from the children of the object pointed to by r_1. To this end, it first checks that $\mu(r_0)$ is in fact the parent of $\mu(r_2)$. Then,

```
domain = function(o0, m){
    return o0[@tag] && (m == "removeChild");
}

check = function(o0, m1, o2, σ0, σ1, σ2){
    var i = $index(o0, o2);
    return [$check(σ0 ⊔ σ1 ⊔ σ2 ≤ o0[$shadow(i)]);]†
}

label = function(ret, o0, m1, o2, σ0, σ1, σ2){
    var j = $index(o0, o2);
    while(j < o0.length − 1){
        [o0[$shadow(j)] = o0[$shadow(j + 1)];]††
    }
    return [σ0 ⊔ σ1 ⊔ σ2]†††;
}
```

Figure 4 IFlow Signature of $\Downarrow_{\mathsf{rem}}$.

the object $\mu(r_0)$ is updated by shifting by -1 all the indexes equal to or higher than i (the index of the object being removed) and by removing its index n. The levels of the indexes of the right siblings of the node to remove are accordingly shifted by -1. The constraint of the rule prevents a program from removing in a high context a node that was inserted in a low context. Function $\mathcal{R}_{\#\mathsf{Children}}$ receives a memory μ as input and outputs a binary relation such that if $\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu)$, then the DOM node pointed to by r has n children (with indexes $0, \ldots, n-1$).

REMOVECHILD

$$\frac{\begin{array}{c}\mu(r_0)(i) = r_2 \qquad \langle r_0, n+1\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \mathsf{dom}(o_0) = \mathsf{dom}(\gamma_0) = \mathsf{dom}(\mu(r_0))\backslash\{n\} \\ \forall_{0\le j<i}\ .\ o_0(j) = \mu(r_0)(j) \qquad \forall_{i\le j<n}\ .\ o_0(j) = \mu(r_0)(j+1) \qquad o_0(@tag) = \mu(r_0)(@tag) \\ \forall_{0\le j<i}\ .\ \gamma_0(j) = \Gamma(r_0)(j) \qquad \forall_{i\le j<n}\ .\ \boxed{\gamma_0(j) = \Gamma(r_0)(j+1)}^{\dagger\dagger} \qquad \gamma_0(@tag) = \Gamma(r_0)(@tag) \\ \mu' = \mu\,[r_0 \mapsto o_0] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \gamma_0] \qquad \boxed{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Gamma(r_0)(i)}^{\dagger}\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, m_1, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \ \Downarrow_{\mathsf{rem}}\ \langle \mu', \Gamma', \Sigma, r_2, \boxed{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2}^{\dagger\dagger\dagger}\rangle}$$

In order for DOM API relations to be added to the semantics, one has to add them to the API register. Hence, we assume that the $\mathcal{R}_{\mathsf{API}}$ extends the API register given in Figure 3. The following lemma validates the hypotheses of the security theorem (Theorem 8) for $\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}$, allowing us to conclude that the extension of a noninterferent JavaScript monitor with the DOM API relations here defined is noninterferent.

▶ **Lemma 14** (Confinement and Noninterference for the DOM API). $\mathbf{Conf}(\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}) \wedge \mathbf{NI}(\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}})$

Figure 4 presents a possible IFlow signature for the API relation $\Downarrow_{\mathsf{rem}}$, which makes use of the following runtime functions: (1) $\$check$ diverges if its argument is different from *true* and returns *true* otherwise; (2) $\$shadow$ receives as input a property name and outputs the name of the corresponding shadow property; and (3) $\$index$ outputs the index of its second argument in the list of children of its first argument. The labeled boxes in the API relation rule and in the code of the IFlow signature are intended to emphasize the correspondence between the two.

6 Conclusion

In summary, we have proposed a methodology for extending arbitrary monitored JavaScript semantics with secure APIs, which allows to prove the security of the extended monitor in a modular way. As a case study, we extend the inlining compiler of [10] with a fragment of the DOM Core Level 1 API. Further related technical developments, as well as an implementation that includes the IFlow signatures of the APIs studied in the paper, can be found in [20].

This work has been partially supported by the EPSRC Grant Reference EP/H008373/1.

References

1. The 5.1th edition of ECMA 262 June 2011. ECMAScript Language Specification. Technical report, ECMA, 2011.
2. A. Almeida-Matos, J. Fragoso Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *TGC*, 2014.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.
4. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.
5. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.
6. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW*, 2002.
7. N. Bielova. Survey on javascript security policies and their enforcement mechanisms in a web browser. *Special Issue on Automated Specification and Verification of Web Systems of JLAP*, 2013.
8. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *CSF*, 2010.
9. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.
10. J. Fragoso Santos and T. Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of Javascript. In *SEC*, 2014.
11. P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. Dom: Towards a formal specification. In *PLAN-X*, 2008.
12. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
13. A. Guha, B. Lerner, J. Gibbs Politz, and S. Krishnamurthi. Web api verification: Results and challenges. 2012.
14. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*.
15. D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*, 2012.
16. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 2012.
17. W3C Recommendation. DOM: Document Object Model (DOM). Technical report, W3C, 2005.
18. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
19. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
20. José Fragoso Santos and Tamara Rezk. Information flow monitor-inlining compiler. http://www-sop.inria.fr/indes/ifJS/.
21. A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *SP*, 2011.
22. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS*, 2006.
23. Deepak Garg Vineet Rajani, Abhishek Bichhawat and Christian Hammer. Information Flow control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015. to appear.

A DOM API Relations

This appendix describes the labeled API relations with which we extend the JavaScript semantics for interaction with DOM objects.

A.1 Auxiliary Semantic Functions

Our specification of the DOM API relations makes use of the following semantic functions:

- $\mathcal{R}_{\#\mathsf{Children}}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathbb{N}$, such that if $\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu)$, then the DOM node pointed to by r has n children (meaning that it defines the indexes $0, \cdots, n-1$).
- $\mathcal{R}_{\mathsf{Ancestor}}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{\mathsf{Ancestor}}(\mu)$, then the DOM node pointed to by r_0 is an ancestor of the DOM node pointed to by r_1 in the DOM forest stored in μ.
- $\mathcal{R}_{\mathsf{Parent}}$ receives a memory μ as input and outputs a relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu)$, then the DOM node pointed to by r_0 is the *parent* of the DOM node pointed to by r_1 (meaning that there is an index i such that $\mu(r_0)(i) = r_1$).
- $\mathcal{O}rphan$ receives a memory μ as input and ouputs a set of references, such that if $r \in \mathcal{O}rphan(\mu)$, then the DOM node pointed to by r is an *orphan* node, that is, it does not have a parent in the DOM forest stored in μ (meaning that it is the root of a dangling tree).

A.2 DOM API Relations - Invariants

Indexes Invariant. When appending a new node to a given node, its index depends on the indexes of the nodes that were already appended. Analogously, when removing a node, the new indexes of its right siblings depend on the index of the node that is to be removed. To tackle this problem, we specify the semantic relations corresponding to the DOM methods *removeChild* and *appendChild* in such a way that, for every DOM node, the level of the property through which it is accessed is always lower than or equal to the levels of the properties corresponding to its right siblings. We refer to this invariant as the *indexes invariant.*

Parent Node Invariant. In the formal model, a DOM object does not define a property pointing to its parent. However, the API relations are specified in such a way that the structure security level of a DOM node works as the level of a "ghost" property pointing to its parent node. Hence, if the structure of a DOM object is observable, it also means that its parent is also observable.

A.3 DOM API Relations - Specification

In the following, we explain the monitored API rules given in Figure 5. In the specification of each API, when an element of the initial configuration is not used in the premises of the corresponding rule, we denote it by _.

- [CREATEELEMENT] The API relation $\Downarrow_{\mathsf{cre}}$ creates a new DOM Element node with tag m and binds a free reference r to it. The structure security level of the newly created node as well as the level of its property @*tag* are both set to $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2$ in order to verify the confinement property (Definition 5).

createElement

$$\frac{r \notin \mathsf{dom}(\mu) \qquad \mu' = \mu\,[r \mapsto [@tag \mapsto m]] \qquad \Gamma' = \Gamma\,[r \mapsto [@tag \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]] \qquad \Sigma' = \Sigma\,[r \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]}{\langle \mu, \Gamma, \Sigma, \langle \#doc, \_, m\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{cre}}\; \langle \mu', \Gamma', \Sigma', r, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

removeChild

$$\frac{\begin{array}{c}\mu(r_0)(i) = r_2 \qquad \langle r_0, n+1\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \mathsf{dom}(o_0) = \mathsf{dom}(\gamma_0) = \mathsf{dom}(\mu(r_0))\backslash\{n\} \\ \forall_{0\le j<i}\cdot o_0(j) = \mu(r_0)(j) \qquad \forall_{i\le j<n}\cdot o_0(j) = \mu(r_0)(j+1) \qquad o_0(@tag) = \mu(r_0)(@tag) \\ \forall_{0\le j<i}\cdot \gamma_0(j) = \Gamma(r_0)(j) \qquad \forall_{i\le j<n}\cdot \gamma_0(j) = \Gamma(r_0)(j+1) \qquad \gamma_0(@tag) = \Gamma(r_0)(@tag) \\ \mu' = \mu\,[r_0 \mapsto o_0] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \gamma_0] \qquad \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Gamma(r_0)(i)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{rem}}\; \langle \mu', \Gamma', \Sigma, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

appendChild - orphan node

$$\frac{\begin{array}{c}\langle r_2, r\rangle \notin \mathcal{R}_{\mathsf{Ancestor}}(\mu) \qquad r_2 \in \mathcal{O}rphan(\mu) \qquad \langle r_0, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \\ \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Sigma(r_0) \sqcap \Sigma(r_2) \\ \mu' = \mu\,[r_0 \mapsto \mu(r_0)\,[n \mapsto r_2]] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \Gamma(r_0)\,[n \mapsto \Sigma(r_0) \sqcup \Sigma(r_2)]]\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu', \Gamma', \Sigma, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

appendChild - non-orphan node

$$\frac{\begin{array}{c}\langle r_p, r_2\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \qquad \langle \mu, \Gamma, \Sigma, \langle r_p, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{rem}}\; \langle \mu', \Gamma', \Sigma', \_, \_\rangle \\ \langle \mu', \Gamma', \Sigma', \langle r_0, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu'', \Gamma'', \Sigma'', \_, \_\rangle \qquad \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Sigma(r_2)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu'', \Gamma'', \Sigma'', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

length

$$\frac{\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r)}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{len}}\; \langle \mu, \Gamma, \Sigma, n, \sigma\rangle}$$

parentNode

$$\frac{v = \begin{cases} r_p & \text{if } \langle r_p, r\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \\ \mathit{undefined} & \text{otherwise}\end{cases} \qquad \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r_2)}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{par}}\; \langle \mu, \Gamma, \Sigma, v, \sigma\rangle}$$

index

$$\frac{\langle v, \sigma\rangle = \begin{cases} \langle \mu(r)(i), \Gamma(r)(i)\rangle & \text{if } i \in \mathsf{dom}(\mu(r)) \\ \langle \mathit{undefined}, \Sigma(r)\rangle & \text{otherwise}\end{cases}}{\langle \mu, \Gamma, \Sigma, \langle r, i\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{ind}}\; \langle \mu, \Gamma, \Sigma, v, \sigma_0 \sqcup \sigma_1 \sqcup \sigma)\rangle}$$

nextSibling

$$\frac{\begin{array}{c}\langle r_p, r\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \qquad \langle r_p, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \\ \langle v_i, \sigma_i\rangle = \begin{cases} \langle \mu(r_p)(i+1), \Gamma(r_p)(i+1)\rangle & \text{if } i+1 < n \\ \langle \mathit{undefined}, \Gamma(r_p)\rangle & \text{otherwise}\end{cases} \\ \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_i \sqcup \Sigma(r)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{sib}}\; \langle \mu, \Gamma, \Sigma, v_i, \sigma\rangle}$$

Figure 5 DOM API Relations

- [removeChild] The API relation $\Downarrow_{\mathsf{rem}}$ removes the node pointed to by r_2 from the list of children of the node pointed to by r_0, after checking that $\mu(r_0)$ is in fact the parent of $\mu(r_2)$. The object $\mu(r_0)$ is updated by shifting by -1 all the indexes equal to or higher than i (the index of the object being removed) and by removing index n. The levels of

the indexes of the right siblings of the node to remove are accordingly shifted by -1. The constraint of the rule prevents a program from removing in a high context a node that was inserted in a low context (see Example 13).

- [APPENDCHILD] The API relation $\Downarrow_{\mathsf{app}}$ has two different behaviors depending on the fact that the node pointed to by r_2 is or is not an orphan node. If the node pointed to by r_2 is an orphan node, the behavior of $\Downarrow_{\mathsf{app}}$ is the following: (1) it first checks that the node to append ($\mu(r_2)$) is not an ancestor of the node to which it is to be appended ($\mu(r_0)$); (2) it creates a new property n in $\mu(r_0)$ and sets it to point to $\mu(r_2)$ (where n is the previous number of children of $\mu(r_0)$); (3) the level of the new index property n is set to the least upper bound on the levels of the arguments and the level of its new left sibling provided that it exists (in order to enforce the *Indexes Invariant*); (4) the least upper bound on the level of the arguments must be equal to or lower than the structure security level of $\mu(r_0)$ because adding an index to a node changes its domain; (5) the least upper bound on the level of the arguments must be equal to or lower than the structure security level of $\mu(r_2)$ (in order to enforce the *Parent Node Invariant*). If the node pointed to by r_2 is not an orphan node, the behavior of $\Downarrow_{\mathsf{app}}$ is the following: (1) it removes $\mu(r_2)$ from the list of children of its current parent (using the $\Downarrow_{\mathsf{rem}}$ API relation); (2) the API relation $\Downarrow_{\mathsf{app}}$ calls itself recursively.
- [LENGTH] The API relation $\Downarrow_{\mathsf{len}}$ evaluates to the number of children of $\mu(r)$. The reading effect of a call to this API must be higher than or equal to the structure security level of $\mu(r)$ because it leaks information about the domain of $\mu(r)$. Concretely, by calling this API relation, one finds out which are the index properties that the node defines.
- [PARENTNODE] The API relation $\Downarrow_{\mathsf{par}}$ evaluates either to the reference that points to the parent of $\mu(r)$, or to *undefined* if $\mu(r)$ is an orphan node. The reading effect of a call to this API is higher than or equal to the structure security level of $\mu(r)$ because it acts as the level of a "ghost" property pointing to the corresponding parent node.
- [INDEX] The API relation $\Downarrow_{\mathsf{ind}}$ evaluates to the ith child of $\mu(r)$. If $\mu(r)$ has less than $i + 1$ children the call to this API returns *undefined*. Besides the security levels of the arguments, the reading effect of a call to this API must take into account either the security level associated with index i (provided that it is defined), or the structure security level of $\mu(r)$ (if it does not exist).
- [NEXTSIBLING] The API relation $\Downarrow_{\mathsf{sib}}$ evaluates either to the reference that points to the right sibling of $\mu(r)$, or to *undefined* if $\mu(r)$ does not have a right sibling. In the former case, the reading effect of a call to this API is higher than or equal to the security level associated with the index pointing to the right sibling, whereas in the latter case it must be higher than or equal to the structure security level of the parent node of $\mu(r)$.