Modular Extensions of Security Monitors for Web APIs: The DOM API Case Study

José Fragoso Santos and Tamara Rezk

No Institute Given

Abstract. Client-side JavaScript programs often interact with the web page on which they are included, as well as the browser itself, through APIs such as the DOM API, the XMLHttpRequest API, and the W3C Geolocation API. The continuous emergence and heterogeneity of different APIs renders the problem of reasoning precisely about JavaScript security particularly challenging. To tackle this problem, we propose a methodology for extending sound JavaScript information flow monitors. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, proving that a monitor is noninterferent after extending it with a new API only requires the proof that the API is itself noninterferent.
We show how to extend an information flow monitor-inlining compiler so that it takes into account the invocation of arbitrary APIs and we provide an implementation of such a compiler.
Using the proposed methodology, we present a group of monitor extensions for handling a fragment of the DOM Core Level 1 API for which we have studied information leaks not explored in previous work.

1 Introduction

Web application designers as well as its users are interested in isolation properties for trusted JavaScript code. In particular, they want to prevent information flows from confidential/untrusted resources to public/trusted ones. The noninterference property [13] provides the mathematical foundations for reasoning precisely about isolation.

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 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 that is often far from trivial. 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 [17]. 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.

We model a fragment of the DOM Core Level 1 API that includes the creation, deletion, and insertion of DOM Element nodes, as well as flexible and previously unmodeled ways of traversing the DOM forest. We show that the presented DOM monitor extensions are compliant with the proposed noninterference property exposing new security leaks related to tree-like structures that complement those studied in [25].

We show how to extend an inforation 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. The compiler is available online [1] and the user can trivially 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.

The contributions of the paper are the following: (1) a methodology for extending JavaScript monitors with API monitoring (Sect. 3.2), (2) an information flow monitor that handles an important subset of the DOM API (Sect. 5), (3) an implementation [1] of an information flow monitor-inlining compiler (Sect. 4) that handles (3).

1.1 Related work

Security Policies for Client-Side JavaScript Programs. There are plenty of mechanisms that serve the purpose of preventing security violations spawned by client-side JavaScript code. This is, however, a very hard goal to attain due to the complexity of JavaScript semantics, as well as the numerous ways in which JavaScript programs can interact with their runtime environment. For instance, the Facebook Javascript Subset [14] that was intended to isolate trusted code

from code coming from untrusted sources, was shown to be breachable by Maffeis and Taly [20]. Analogously, Politz et al. [22] have shown that the implementation of Yahoo ADsafe [12], a sandboxing mechanism for isolating trusted JavaScript code, is compromised by several bugs that can be exploited to encode security violations. We refer the reader to [8] for a recent survey on enforcement mechanisms for securing client-side JavaScript programs, while focusing here on the most closely related work. In contrast to other more *adhoc* isolation mechanisms, noninterference is an expressive and elegant property that defines secure information flow. The dynamic nature of JavaScript makes it a difficult target for static analysis. Therefore, we turn our attention towards dynamic mechanisms for enforcing noninterference and refer the reader to [26] for a survey of the field.

Information Flow Monitors and Monitor-Inlining Compilers. Flow-sensitive monitors for enforcing noninterference can be broadly divided into two classes: those that are purely dynamic, such as [3], [4], and [5], and those (commonly referred to as *hybrid monitors*) that mix runtime monitoring with static analysis, such as [30], [16], and [27]. In contrast to hybrid monitors, which rely on static analysis to reason about implicit flows that arise due to untaken execution paths, purely dynamic monitors do not rely on any kind of static analysis. Instead, the authors of [3], [4], and [5] propose three alternative strategies in designing sound purely dynamic information flow monitors. The *no-sensitive-upgrade* strategy forbids the update of public resources inside private contexts. The *permissive-upgrade* strategy allows sensitive upgrades to take place, but marks the resources upgraded in sensitive contexts and forbids the program to branch depending on the content of these resources. Finally, the *multiple facet* strategy surpasses the limitations of the first two (which can potentially abort the execution of secure programs) by the use of multiple faceted values. The intuition behind this strategy is that values must appear differently to observers at different security levels. Therefore, the security monitor simulates multiple executions for different security levels. Russo et. al [24] show (for a WHILE language) that purely dynamic monitors always reject executions that would have been accepted using static enforcement mechanisms. In this work, we show how to extend an arbitrary information flow monitor without making any assumption on its particular design. However, the monitor extensions that illustrate the applicability of our methodology follow the no-sensitive-upgrade discipline of [3]. Chudnov and Naumann [11] propose the inlining of a hybrid information flow monitor for a simple imperative language. Simultaneously, Magazinius et al. [21] also propose a monitor inlining transformation for an imperative language, with the novel feature of performing inlining on the fly to allow for dynamic code evaluation. In the implementation, we apply the techniques presented in [21] and [11]. Hedin and Sabelfeld [18] are the first to propose a runtime monitor for enforcing noninterference for JavaScript. The technique that we present for extending security monitors can be trivially applied to this monitor, which is a purely dynamic monitor and therefore liable to the limitations described in [24]. To account for these limitations, Birgisson et. al [9] show how to use tests in order to boost the permissiveness of [18]. Each time a security error arises during a

test, the program is modified with an annotation that prevents the same error from reoccurring. Despite targeting JavaScript, these two works, as our own, do not model the reactive aspect of client-side web applications. Bohannon et al. [10] present a definiton of noninterference for reactive programs such as web scripts as well as a runtime monitor for enforcing it.

Securing Web APIs. Taly et al. [28] study the problem of API confinement. They provide a static analysis designed to formally verify whether, when integrating the code of an API in an arbitrary page, the integrator code cannot interact with the API and cause it to leak its confidential resources. They use, however, a more restrictive notion of a web API than the one used in this work, in the sense that they use the term API only to refer to JavaScript libraries whose code is explicitly included by the programmer and, therefore, available for either runtime or static analysis.

The DOM API - Formal Semantics and Secure Information Flow Enforcement. Russo et al. [25] 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; instead, program configurations include the node on which the program is operating. They point out this feature of their language as the biggest difference between their semantics and JavaScript DOM operations because, in JavaScript, programs are allowed to store in memory references to arbitrary nodes in the DOM forest. In fact,This feature of their semantics allows them to avoid reasoning about complex aspects related to information flows triggered by JavaScript DOM operations; for instance, the fact that a program can append to the DOM forest a node that is already part of it. Gardner et al. [15] propose a compositional and concise formal specification of the DOM. that they call Minimal DOM. They further show that their semantics has no redundancy. and that it is sufficient to describe the structural kernel of DOM Core Level 1, meaning that the semantics of all the other unmodeled commands can be obtained from that of the modeled ones. Additionally, they apply local reasoning based on Separation Logic and prove invariant properties of simple JavaScript programs. We choose to follow the DOM semantics given in [15]. However, our DOM semantics is different from theirs in that we model DOM objects in the same way we model JavaScript objects. Furthermore, our main concern lies with the tracking of information flow in the DOM and not with the modeling of the semantics itself.

2 Noninterference for JavaScript

2.1 JavaScript Memory Model

A memory $\mu : \mathcal{R}ef \mapsto \mathcal{S}tr \mapsto \mathcal{P}rim \cup \mathcal{R}ef \cup \mathcal{F}_\lambda$ is a mapping from references to objects [2]. We model objects as partial functions mapping strings taken from set $\mathcal{S}tr$ to values taken from $\mathcal{P}rim \cup \mathcal{R}ef \cup \mathcal{F}_\lambda$, where $\mathcal{P}rim$ is the set of primitive values, $\mathcal{R}ef$ the set of references, and $\mathcal{F}_\lambda$ the set of parsed function literals. $\mathcal{P}rim$ includes strings, numbers, and booleans, as well as, two special values *null* and *undefined*. References can be viewed as pointers to objects, in the

sense that every expression that creates an object in memory yields a free non-deterministically chosen reference that points to it. The strings in the domain of an object are called its *properties*. Some properties cannot be changed by the program. 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.

The definitions given in this section make use of a small-step semantics of JavaScript $\rightarrow_{JS}$ that relates configurations of the form $\langle \mu, s \rangle$, where μ is a JavaScript memory and s the program to evaluate. The execution of a program s on a memory μ is said to terminate in $\rightarrow_{JS}$ *if and only if* there is a memory μ' and a value $v \in \mathcal{P}rim \cup \mathcal{R}ef$ such that $\langle \mu, s \rangle \rightarrow_{JS}^{*} \langle \mu', v \rangle$, where $\rightarrow_{JS}^{*}$ denotes the reflexive-transitive closure of $\rightarrow_{JS}$. In the following, we assume that the binding of variables is modeled using *scope objects* and that the evaluation of a function literal triggers the creation of a function object that stores its parsed counterpart and a reference to the scope object associated with the environment in which it was evaluated [19]. In particular, we assume the existence of a relation $\mathcal{R}_{Scope}$ that models the variable look-up procedure. If $\langle \mu, x \rangle \; \mathcal{R}_{Scope} \; v$ then v is the value associated with variable x in memory μ. Observe that unlike the authors of [19], we do not include a reference to the *active scope object* (the one that models the current scope) in program configurations. We choose to do so in order to keep configurations as simple as possible. One can design a small-step semantics of JavaScript that fullfills this requirement by storing the reference pointing to the active scope object in a fixed internal property of the global object.

We use the *notation*: (1) $[p_0 \mapsto v_0, \cdots, p_n \mapsto v_n]$ for the partial function that maps p_0 to v_0, ..., and p_n to v_n resp., (2) $f\,[p_0 \mapsto v_0, \cdots, p_n \mapsto v_n]$ for the function that coincides with f everywhere except in p_0, ..., p_n, which are otherwise mapped to v_0, ..., v_n resp., (3) $f|_P$ for the restriction of f to P (provided it is included in its domain), and (4) $f(r)(p)$ for $(f(r))(p)$, that is, the application of the image of r by f (which is assumed to be a function) to p.

2.2 Noninterference

Information flow policies, such as noninterference, are usually specified by labeling the observable resources of a program, such as properties and variables, with security levels taken from a given security lattice. In the following, we make use of a lattice $\mathcal{L}$ of security levels. We denote by $\bot$ the bottom element of the lattice, by $\leq$ its order relation, and by $\sigma_0 \sqcup \sigma_1$ the least upper bound between security levels σ_0 and σ_1. In the examples, we use the lattice $\mathcal{L} = \{H, L\}$ with $L \leq H$; meaning that resources labeled with level *low* L are less confidential than resources labeled with level *high* H. Hence, after the execution of a program, resources labeled with L cannot depend on resources originally labeled with H, since that would entail a security violation.

Labeling variables or other resources in a program is frequently done statically via a mapping from statically referred resources to security levels. In JavaScript, resources are dynamically created which makes it infeasible to refer to them precisely at the static level.

Example 1 (Dynamic Resource Allocation in JavaScript). Consider the program $x = \{\}; x[f()] = 1$ where function f returns a string m obtained by concatenating arbitrary user input. Here, a reference to an object o is stored in variable x and then m is added to o as a property. However, property m cannot be precisely labeled at static time since its name is not known.

Hence, we consider security labelings that map object properties to security levels. Formally, a security labeling is modeled as a pair $\langle \Gamma, \Sigma \rangle$, where $\Gamma : \mathcal{R}ef \mapsto \mathcal{S}tr \mapsto \mathcal{L}$ maps references in $\mathcal{R}ef$ and properties in $\mathcal{S}tr$ to security levels and $\Sigma : \mathcal{R}ef \mapsto \mathcal{L}$ maps references to security levels. Therefore, given an object o pointed to by a reference r, $\Gamma(r)(p)$, if defined, is the security level associated with o's property p and $\Sigma(r)$, if defined, is the *structure security level* of o [18]. We say that memory μ is well-labeled by $\langle \Gamma, \Sigma \rangle$ if $dom(\Gamma) = dom(\Sigma) \subseteq dom(\mu)$ and for every reference $r \in dom(\Gamma)$, $dom(\Gamma(r)) \subseteq dom(\mu(r))$.

The need for the structure security level arises from the fact that, in JavaScript, looking-up the value of an undefined property does not yield an error, but, instead, *undefined*. Hence, one can encode information flows through the domain of an object by creating (or deleting) properties in unobservable contexts and afterwards testing whether the corresponding look-up yields *undefined*. The structure security level of an object is, therefore, meant to model the security level of its domain.

Example 2 (The need for the structure security level). Consider the following program where object o does not initially have any properties: $\mathsf{if}(h)\ \{o.p = 0\}; l = o.p$. Intuitively, the domain of o depends on the high variable h, because property "p" is created in a branch of the IF statement that is taken depending on the value of h. Therefore, variable l depends on a high variable as well, since its final value will be 0 when $h \notin \{false, 0, null, undefined\}$ and *undefined* otherwise.

In contrast to previous works [3, 18] on dynamic information flow analysis, we choose to separate the enforcement mechanism from the definition of noninterference. This allows us to distinguish the class of secure programs that are rejected by the enforcement mechanism (due to over-conservative constraints) from the class of insecure programs. In addition, since the proposed noninterference definition is independent from the enforcement mechanism, not only can it be compared with other security properties, but it can also be used as the soundness criterion for other security analyses. The definition of noninterference depends on a notion of indistinguishability between memories that models the power of an attacker that can only observe resources up to a security level σ, called *low-equality* and denoted by $\approx_{\beta,\sigma}$. Informally, if two labeled memories are low-equal at level σ, then they coincide in the resources labeled with levels $\leq \sigma$. Hence, an attacker that can only observe resources up to level σ, cannot distinguish them. To account for the non-deterministic behavior of the memory allocator, which can yield different references that point to the same resource in different executions of the same program, we rely on a partial injective function $\beta : \mathcal{R}ef \to \mathcal{R}ef$ [6]. For every two executions, there is a function β relating the

observable references that point to the same resource. In the following, we always assume β to be injective. A partial function $\beta : \mathcal{R}ef \rightarrow \mathcal{R}ef$ is said to be proper if it is injective and $\beta(\#glob) = \#glob$ and $\beta(\#protObj) = \#protObj$.The low-equality definition relies on a relation on observable values, named β-equality, and denoted $\sim_\beta$. *β-Equality:* two objects are β-equal if they have the same domain and all their corresponding properties are β-equal, primitive values and parsed function are β-equal if equal, and two references r_0 and r_1 are β-equal if the latter is the image of the former. In the following, given a property labeling Γ, a reference r, and a security level σ, we denote by $\Gamma(r)|_\sigma$, the set of observable properties in $\Gamma(r)$ at level σ – $\{p \mid \Gamma(r)(p) \leq \sigma\}$. *Low-equality:* 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 $r_0, r_1 \in dom(\beta)$, such that $r_1 = \beta(r_0)$, the following holds:

1. The observable domains coincide: $P = \Gamma_0(r_0)|_\sigma = \Gamma_1(r_1)|_\sigma$;
2. The objects coincide in the observable domain: $\mu_0(r_0)|_P \sim_\beta \mu_1(r_1)|_P$;
3. If the structure security level of one of the two objects is observable, then their domains entirely coincide: $\big(\Sigma_0(r_0) \leq \sigma \vee \Sigma_1(r_1) \leq \sigma\big) \Rightarrow dom(\mu_0(r_0)) = dom(\mu_1(r_1))$.

A program is said to be noninterferent if, whenever it is executed on two low-equal memories, produces two low-equal memories and two β-equal values. Hence, an attacker that can only see the low resources of a program cannot infer anything about its high resources.

3 Modular Extensions for JavaScript Monitors

In this section, we give the definition of *noninterferent JavaScript security monitor* and we show how to extend such a monitor so that it (1) takes into account the invocation of web APIs and (2) remains noninterferent.

3.1 Noninterferent JavaScript Monitors

The proposed mechanism for extending security monitors expects as input a monitored small-step semantics that relates configurations of the general form $\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle$, where (1) μ is the current memory, (2) s the current program to execute, (3) $\overrightarrow{\sigma}_{pc}$ a sequence of security levels matching the reading effects of the expressions on which the original program branched to reach to the current context, (4) $\langle \Gamma, \Sigma \rangle$ the current labeling, and (5) $\overrightarrow{\sigma}$ a sequence of

security levels matching the reading effects of the subexpressions of the expression being computed, which were already computed. The *reading effect* [26] 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, $\sqcup\overrightarrow{\sigma}_{pc}$. We use the notation (1) $\sqcup\overrightarrow{\sigma}$ for the least upper bound on the levels in $\overrightarrow{\sigma}$, (2) $\lfloor\overrightarrow{\sigma}\rfloor_n$ for the sequence containing the last n elements of $\overrightarrow{\sigma}$, (3) $|\overrightarrow{\sigma}|$ for the number of elements of $\overrightarrow{\sigma}$, (4) $\cdot$ for the concatenation of sequences, (5) ϵ for the empty sequence, and (6) $\beta(\overrightarrow{v})$ for the sequence of values that is obtained from $\overrightarrow{v}$ by applying to each one of the elements of $\overrightarrow{v}$ either function β (provided it is in its domain), or the identity function. In the following, we extend the definition of low-equality to sequences of labeled values and to program configurations.

Informally, two sequences of labeled values are low-equal with respect to a given security level σ, 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. Hence, two sequences of values $\langle v_0, \cdots, v_n\rangle$ and $\langle v'_0, \cdots, v'_k\rangle$, respectively associated with two sequences of levels $\langle\sigma_0, \cdots, \sigma_n\rangle$ and $\langle\sigma'_0, \cdots, \sigma'_k\rangle$ are low-equal w.r.t. a security level σ and a function β, written $\langle v_0, \cdots, v_n\rangle, \langle\sigma_0, \cdots, \sigma_n\rangle \approx_{\beta,\sigma} \langle v'_0, \cdots, v'_k\rangle, \langle\sigma'_0, \cdots, \sigma'_k\rangle$, if, letting $j = \mathsf{min}(n, k)$, either $n = k = 0$, or the following holds: (1) $\forall i \leq j \cdot (\sigma_i \leq \sigma \vee \sigma'_i \leq \sigma) \Rightarrow v_i \sim_\beta v'_i$ and (2) $(\forall j < i \leq n \cdot \sigma_i \not\leq \sigma) \ \wedge \ (\forall j < i \leq k \cdot \sigma'_i \not\leq \sigma)$.

Two program configurations $\langle\mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma}\rangle$ and $\langle\mu', s', \overrightarrow{\sigma}'_{pc}, \Gamma', \Sigma', \overrightarrow{\sigma}'\rangle$ are said to be *low-equal* w.r.t. a security level σ and a function β, written $\langle\mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma}\rangle \approx_{\beta,\sigma} \langle\mu', s', \overrightarrow{\sigma}'_{pc}, \Gamma', \Sigma', \overrightarrow{\sigma}'\rangle$, if $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \sigma'$ and $s, \overrightarrow{\sigma}_{pc}, \overrightarrow{\sigma} \approx_{\beta,\sigma} s', \overrightarrow{\sigma}'_{pc}, \overrightarrow{\sigma}'$. The low-equality between programs annotated with two sequences of security levels relates the intermediate states of the execution of the same original program in two low-equal memories.

Example 5 (Low-equal programs). Consider the program $x = y$, an original 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 memories 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: $x = 0, \langle L\rangle, \langle H\rangle \approx_{id,L} x = 1, \langle L\rangle, \langle H\rangle$.

The design of the low-equality for programs is tightly related with the design of the monitor. Therefore, we do not include it in the paper. Instead, we only state the properties that it must verify for the results presented in this section to hold:

1. Given two expression redexes (expressions that only have values as subexpressions) e and e' and two pairs of sequences of values $\overrightarrow{\sigma}_{pc}$ and $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'_{pc}$ and $\overrightarrow{\sigma}'$; if $s, \overrightarrow{\sigma}_{pc}, \overrightarrow{\sigma} \not\approx_{\beta,\sigma} s', \overrightarrow{\sigma}'_{pc}, \overrightarrow{\sigma}'$, for a security level σ and a function β, then $(\sqcup\lfloor\overrightarrow{\sigma}\rfloor_n) \sqcap (\sqcup\lfloor\overrightarrow{\sigma}'\rfloor_k) \not\leq \sigma$ (where n and k correspond to the number of subexpressions of e and e' resp.).

2. For any two values v_0 and v_1 respectively associated with two security level σ_0 and σ_1, security level σ, function β, and two sequences of security levels $\overrightarrow{\sigma}^0_{pc}$ and $\overrightarrow{\sigma}^1_{pc}$, if $\langle v_0\rangle, \langle\sigma_0\rangle \approx_{\beta,\sigma} \langle v_1\rangle, \langle\sigma_1\rangle$, then: $v_0, \overrightarrow{\sigma}^0_{pc}, \langle\sigma_0\rangle \approx_{\beta,\sigma} v_1, \overrightarrow{\sigma}^1_{pc}, \langle\sigma_1\rangle$.

Definition 3 (Monitor Noninterference). *A monitor $\rightarrow_{IF}$ is said to be noninterferent, written $\mathbf{NI}_{monitor}(\rightarrow_{IF})$, if and only if for every program s, memory μ, and labeling $\langle\Gamma, \Sigma\rangle$, such that μ is well-labeled by $\langle\Gamma, \Sigma\rangle$, if $\langle\mu, s, \epsilon, \Gamma, \Sigma, \epsilon\rangle \rightarrow^*_{IF} \langle\mu', v', \epsilon, \Gamma', \Sigma', \sigma'\rangle$ for some memory μ', value v', labeling $\langle\Gamma', \Sigma'\rangle$, and level σ', then $\mu \in \mathbf{NI}^{\sigma'}_{exec}(s, \langle\Gamma, \Sigma\rangle, \langle\Gamma', \Sigma'\rangle)$. A monitor $\rightarrow_{IF}$ is said to be step-wise noninterferent, written $\mathbf{NI}^{step}_{monitor}(\rightarrow_{IF})$, if the application of the monitor on two low-equal configurations produces two low-equal configurations.*

3.2 Monitor Extensions

Formally, every API is modeled as a semantic relation, $\Downarrow^{JS}_{API}$ of the form:

$$\langle\mu, \overrightarrow{v}\rangle \ \Downarrow^{JS}_{API} \ \langle\mu', v\rangle$$

Where μ is the memory in which the API is executed, μ' is the memory resulting from the execution of the API, $\overrightarrow{v}$ is the sequence of values corresponding to the arguments of the API invocation, and v is the value to which the API invocation evaluates. Accordingly, a monitored API relation, $\Downarrow_{API}$, has the following form:

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

Where $\langle\Gamma, \Sigma\rangle$ and $\langle\Gamma', \Sigma'\rangle$ are the initial and final labelings resp., $\overrightarrow{\sigma}$ is the sequence of security levels associated with the arguments of the API invocation, and σ is its corresponding reading effect. The remaining elements preserve their original meaning. We ommit the terms monitored/unmonitored when they are clear from the context. We define a (monitored) *API register* as a function that given a memory and a sequence of values, returns a (monitored) API semantic relation.

Figure 1 defines a function $\mathcal{E}$ that, given a (monitored) small-step semantics, $\rightarrow$, produces a new (monitored) small-step semantics $\mathcal{E}(\rightarrow, Intercept, \mathcal{R}_{API})$ that handles the invocation of the APIs in the range of the API register $\mathcal{R}_{API}$

STANDARD EXECUTION

$$\frac{s \notin Intercept \vee (s \in Intercept \wedge \langle \mu, \mathsf{SubExpressions}[[s]] \rangle \notin dom(\mathcal{R}_{API})) \quad \langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \rightarrow \langle \mu', s', \overrightarrow{\sigma}'_{pc}, \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}{\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \ \rightarrow_{API} \ \langle \mu', s', \overrightarrow{\sigma}'_{pc}, \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}$$

API EXECUTION

$$\frac{\begin{array}{c} s \in Intercept \qquad \langle \mu, \mathsf{SubExpressions}[[s]] \rangle \in dom(\mathcal{R}_{API}) \\ |\mathsf{SubExpressions}[[s]]| = n+1 \qquad \overrightarrow{\sigma} = \overrightarrow{\sigma}' \cdot \langle \sigma_0, \cdots, \sigma_n \rangle \qquad \Downarrow_{API} = \mathcal{R}_{API}(\mu, \mathsf{SubExpressions}[[s]]) \\ \langle \mu, \Gamma, \Sigma, Intercept(s), \langle \sigma_0, \cdots, \sigma_n \rangle \rangle \ \Downarrow_{API} \ \langle \mu', \Gamma', \Sigma', v', \sigma' \rangle \end{array}}{\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \ \rightarrow_{API} \ \langle \mu', v', \overrightarrow{\sigma}_{pc}, \Gamma', \Sigma', \overrightarrow{\sigma}' \cdot \sigma' \rangle}$$

Fig. 1. $\mathcal{E}(\rightarrow, Intercept, \mathcal{R}_{API}) = \rightarrow_{API}$

triggered by statements which are included in the set of *intercepting points* – $Intercept$. When $\mathcal{E}$ receives as input a monitored small-step semantics and a monitored API register, it generates a new monitored small-step semantics. Informally, a statement can trigger the evaluation of an API in the new small-step semantics $\mathcal{E}(\rightarrow, Intercept, \mathcal{R}_{API})$ if: (1) it is an intercepting point (that is, $s \in Intercept$) and (2) the sequence of values to which its subexpressions evaluate is in the domain of $\mathcal{R}_{API}$, in which case their image by $\mathcal{R}_{API}$ corresponds to the semantic relation that models the API to execute. We assume that only expression redexes can be used as intercepting points. In the definition of $\mathcal{E}$, Rules [NON-INTERCEPTED PROGRAM CONSTRUCT] and [INTERCEPTED PROGRAM CONSTRUCT - STANDARD EXECUTION] model the case in which the new small-step semantics executes the original rule. Rule [INTERCEPTED PROGRAM CONSTRUCT - API EXECUTION] models the case in which an API is executed. The definition of $\mathcal{E}$ makes use of a syntactic function, SubExpressions, defined on JavaScript expressions, such that $\mathsf{SubExpressions}[[s]]$ corresponds to the sequence comprising all the subexpressions of s in the order by which they are evaluated. We use the notation $\mathcal{E}^*(\rightarrow, Intercept, \mathcal{R}_{API})$, for the reflexive-transitive closure of $\mathcal{E}(\rightarrow, Intercept, \mathcal{R}_{API})$.

Example 7. Consider an API for creating and manipulating priority queues:

- $queueAPI.queue()$: 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.

This API is available for the programmer through the global variable $queueAPI$. For the extended monitor to take into account the methods of this API, one has to specify for each one of them the corresponding API semantic relation. Additionally, one has to define method calls as an interception point of the semantics (by adding method call redexes $Intercept$) and to extend the API Register to make the invocation of these methods trigger the execution of the corresponding APIs. To this end, the Queue API object (the one bound to

variable *queueAPI*) as well as the concrete queue objects are "marked" with a special property (for instance, $\$q$) and the API Register is extended accordingly:

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

where $\Downarrow_{QU}$, $\Downarrow_{PU}$, and $\Downarrow_{PO}$ are the API relations corresponding to each one of the methods of the Queue API.

3.3 Definition of NI($\mathcal{R}_{API}$) for external interfaces

Assuming that the original monitor is noninterferent, for $\mathcal{E}(\rightarrow_{IF}, Intercept, \mathcal{R}_{API})$ to be noninterferent one must impose some constraints on the API relations in the range of $\mathcal{R}_{API}$. Definitions 4 and 5 formalize noninterference for API relations. Definition 4 states that an API relation is *confined* if it only creates/updates resouces 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 4 (Confined API Relation). *An API relation $\Downarrow_{API}$ is confined if for any memory μ well-labeled by $\langle \Gamma, \Sigma \rangle$, any sequence of values $\overrightarrow{v}$ resp. labeled by a sequence of levels $\overrightarrow{\sigma}$, and any security level σ, such that:*

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

*and $\sqcup \overrightarrow{\sigma} \not\leq \sigma$, for some memory μ', labeling $\langle \Gamma', \Sigma' \rangle$, value v', and security level σ'; **then**: $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$ and $\sigma' \not\leq \sigma$.*

Definition 5 states that 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. In order to relate the outputs of the API Register in two low-equal memories, we extend the noninterference definition for 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.

Definition 5 (Noninterferent Api Relation/Register). *An API relation $\Downarrow_{API}$ is said to be noninterferent, written* **NI**$(\Downarrow_{API})$*, if it is confined and for any 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 on $\mathcal{R}ef$ such that $\overrightarrow{v}_0, \overrightarrow{\sigma}_0 \approx_{\beta,\sigma} \overrightarrow{v}_1, \overrightarrow{\sigma}_1$, $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$, and:*

$$\begin{array}{l} \langle \mu_0, \Gamma_0, \Sigma_0, \overrightarrow{v}_0, \overrightarrow{\sigma}_0 \rangle \ \Downarrow_{API} \ \langle \mu'_0, \Gamma'_0, \Sigma'_0, v'_0, \sigma'_0 \rangle \\ \langle \mu_1, \Gamma_1, \Sigma_1, \overrightarrow{v}_1, \overrightarrow{\sigma}_1 \rangle \ \Downarrow_{API} \ \langle \mu'_1, \Gamma'_1, \Sigma'_1, v'_1, \sigma'_1 \rangle \end{array}$$

there is a function β' that extends β such that $\mu'_0, \Gamma'_0, \Sigma'_0 \approx_{\beta',\sigma} \mu'_1, \Gamma'_1, \Sigma'_1$ and $\langle v'_0 \rangle, \langle \sigma'_0 \rangle \approx_{\beta',\sigma} \langle v'_0 \rangle, \langle \sigma'_0 \rangle$. We say that the API Register function ($\mathcal{R}_{API}$) is noninterferent, written **NI**$(\mathcal{R}_{API})$*, if all the API relations in its range are noninterferent and for any 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* $\mathcal{R}_{API}(\mu, \overrightarrow{v}) = \mathcal{R}_{API}(\mu', \beta(\overrightarrow{v}))$*.*

Example 8. Assume that the APIs given in Example 7 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(o0, 0); }
3  q.push(o1, 1); 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 APIs go through (the execution is never blocked), then it must terminate with memories μ'_i labeled by Γ', Σ:

$$\mu_i = \begin{bmatrix} (\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1, \\ (\#glob, h) \mapsto i \end{bmatrix} \quad \mu'_i = \begin{bmatrix} (\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1, \\ (\#glob, h) \mapsto i, (\#glob, l) \mapsto i, \\ (\#glob, q) \mapsto r_q \end{bmatrix}$$

$$\Gamma = \begin{bmatrix} (\#glob, h) \mapsto H, (\#glob, l) \mapsto L, \\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L, \end{bmatrix} \quad \Gamma' = \begin{bmatrix} (\#glob, h) \mapsto H, (\#glob, l) \mapsto H, \\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L, \end{bmatrix}$$

Since the original memories are low-equal, $\mu_0, \Gamma, \Sigma \approx_{\mathsf{id},L} \mu_1, \Gamma, \Sigma$ (where id is the identity function on $\mathcal{R}ef$), we conclude, using the hypothesis that all three API relations are noninterferent, that the memories yielded by the invocation of the API relations in lines 1, 2 (only for the execution that originally maps h to 1), and 3 are also low-equal. Since the value of l clearly depends on the value of h in the execution that originally maps h to 1, we conclude that it is also the case in the execution that originally maps h to 0.

Theorem 1 (Security). *For any monitored semantics* $\rightarrow_{IF}$*, API register* $\mathcal{R}_{API}$ *and set of intercepting points* $Intercept$*, if* $\mathbf{NI}^{step}_{monitor}(\rightarrow_{IF})$ *(hyp.1) and* $\mathbf{NI}(\mathcal{R}_{API})$ *(hyp.2), then* $\mathbf{NI}^{step}_{monitor}(\mathcal{E}(\rightarrow_{IF}, Intercept, \mathcal{R}_{API}))$*.*

Proof. In order to show that $\mathcal{E}(\rightarrow_{IF}, Intercept, \mathcal{R}_{API})$ is step-wise noninterferent we need to show that the low-equality relation for configurations is preserved by every transition of the monitor. Hence, given a security level σ, two memories μ_0 and μ_1, two labelings $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, two programs s_0 and s_1, and four sequences of security levels $\overrightarrow{\sigma}_0$, $\overrightarrow{\sigma}_1$, $\overrightarrow{\sigma}^0_{pc}$, and $\overrightarrow{\sigma}^1_{pc}$ such that:

- $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$ hyp.3
- $s_0, \overrightarrow{\sigma}_0, \overrightarrow{\sigma}^0_{pc} \approx_{\beta,\sigma} s_1, \overrightarrow{\sigma}_1, \overrightarrow{\sigma}^1_{pc}$ hyp.4
- $\langle \mu_0, s_0, \overrightarrow{\sigma}^0_{pc}, \Gamma_0, \Sigma_0, \overrightarrow{\sigma}_0 \rangle \rightarrow_{API} \langle \mu'_0, s'_0, \overrightarrow{\sigma}^{0'}_{pc}, \Gamma'_0, \Sigma'_0, \overrightarrow{\sigma}_{0'} \rangle$ hyp.5
- $\langle \mu_1, s_1, \overrightarrow{\sigma}^1_{pc}, \Gamma_1, \Sigma_1, \overrightarrow{\sigma}_1 \rangle \rightarrow_{API} \langle \mu'_1, s'_1, \overrightarrow{\sigma}^{1'}_{pc}, \Gamma'_1, \Sigma'_1, \overrightarrow{\sigma}_{1'} \rangle$ hyp.6

where we use $\rightarrow_{API}$ for $\mathcal{E}(\rightarrow_{IF}, Intercept, \mathcal{R}_{API})$. We need to show that:

$$\langle \mu'_0, s'_0, \overrightarrow{\sigma}^{0'}_{pc}, \Gamma'_0, \Sigma'_0, \overrightarrow{\sigma}_{0'} \rangle \approx_{\beta',\sigma} \langle \mu'_1, s'_1, \overrightarrow{\sigma}^{1'}_{pc}, \Gamma'_1, \Sigma'_1, \overrightarrow{\sigma}_{1'} \rangle$$

for some β' extending β.

We start by considering the scenario in which $\sqcup \overrightarrow{\sigma}^0_{pc} \not\leq \sigma$ (hyp.7) that corresponds to the case in which execution 0 gives a step in a *high* context. There are four cases to consider:

– I - Both Executions perform a standard transition.
– II - Both Executions perform an API call.
– III - Execution 0 performs a standard transition and Execution 1 performs an API call.
– IV - Execution 0 performs an API call and Execution 1 performs a standard transition.

The proofs of cases III and IV are symmetric. Hence, we ommit case IV.

[Case I.] $s_0 \notin \mathit{Intercept} \vee \langle \mu_0, \mathsf{SubExpressions}[[s_0]]\rangle \notin \mathit{dom}(\mathcal{R}_{API})$ (hyp.8) and $s_1 \notin \mathit{Intercept} \vee \langle \mu_1, \mathsf{SubExpressions}[[s_1]]\rangle \notin \mathit{dom}(\mathcal{R}_{API})$ (hyp.9). It follows that:

– $\langle \mu_0', s_0', \overrightarrow{\sigma}^{0'}_{pc}, \Gamma_0', \Sigma_0', \overrightarrow{\sigma}_{0'}\rangle \approx_{\beta,\sigma} \langle \mu_1', s_1', \overrightarrow{\sigma}^{1'}_{pc}, \Gamma_1', \Sigma_1', \overrightarrow{\sigma}_{1'}\rangle$
(I.1) - hyp.1 + hyp.3 - hyp.6 + hyp.8 + hyp.9

[Case II.] $s_0 \in \mathit{Intercept} \vee \langle \mu_0, \mathsf{SubExpressions}[[s_0]]\rangle \in \mathit{dom}(\mathcal{R}_{API})$ (hyp.8) and $s_1 \in \mathit{Intercept} \vee \langle \mu_1, \mathsf{SubExpressions}[[s_1]]\rangle \in \mathit{dom}(\mathcal{R}_{API})$ (hyp.9). It follows that:

– $\sqcup \overrightarrow{\sigma}^{1}_{pc} \not\leq \sigma$ (II.1) - hyp.4 + hyp.7
– $\mu_0, \Gamma_0, \Sigma_0 \approx_{id,\sigma} \mu_0', \Gamma_0', \Sigma_0'$ (II.2) - hyp.2 + hyp.3 + hyp.5 + hyp.8
– $\mu_1, \Gamma_1, \Sigma_1 \approx_{id,\sigma} \mu_1', \Gamma_1', \Sigma_1'$ (II.3) - hyp.2 + hyp.3 + hyp.6 + hyp.9
– $\mu_0', \Gamma_0', \Sigma_0' \approx_{\beta,\sigma} \mu_1', \Gamma_1', \Sigma_1'$ (II.4) - (II.2) + (II.3) + Lateral Transitivity of $\approx_{\beta,\sigma}$
– $\langle \mu_0', s_0', \overrightarrow{\sigma}^{0'}_{pc}, \Gamma_0', \Sigma_0', \overrightarrow{\sigma}_{0'}\rangle \approx_{\beta,\sigma} \langle \mu_1', s_1', \overrightarrow{\sigma}^{1'}_{pc}, \Gamma_1', \Sigma_1', \overrightarrow{\sigma}_{1'}\rangle$
(III.5) - hyp.4 + hyp.7 + (II.3) + (II.4)

[Case III.] $s_0 \in \mathit{Intercept} \vee \langle \mu_0, \mathsf{SubExpressions}[[s_0]]\rangle \in \mathit{dom}(\mathcal{R}_{API})$ (hyp.8) and $s_1 \notin \mathit{Intercept} \wedge \langle \mu_1, \mathsf{SubExpressions}[[s_1]]\rangle \notin \mathit{dom}(\mathcal{R}_{API})$ (hyp.9). It follows that:

– $\sqcup \overrightarrow{\sigma}^{1}_{pc} \not\leq \sigma$ (III.1) - hyp.4 + hyp.7
– $\mu_0, \Gamma_0, \Sigma_0 \approx_{id,\sigma} \mu_0', \Gamma_0', \Sigma_0'$
(III.2) - hyp.2 + hyp.3 + hyp.5 + hyp.8 *Confinemed Semantic Transition*
– $\mu_1, \Gamma_1, \Sigma_1 \approx_{id,\sigma} \mu_1', \Gamma_1', \Sigma_1'$
(III.3) - hyp.2 + hyp.3 + hyp.6 + hyp.9 *Confinemed API Transition*
– $\mu_0', \Gamma_0', \Sigma_0' \approx_{\beta,\sigma} \mu_1', \Gamma_1', \Sigma_1'$ (III.4) - (III.2) + (III.3) + Lateral Transitivity of $\approx_{\beta,\sigma}$
– $\langle \mu_0', s_0', \overrightarrow{\sigma}^{0'}_{pc}, \Gamma_0', \Sigma_0', \overrightarrow{\sigma}_{0'}\rangle \approx_{\beta,\sigma} \langle \mu_1', s_1', \overrightarrow{\sigma}^{1'}_{pc}, \Gamma_1', \Sigma_1', \overrightarrow{\sigma}_{1'}\rangle$
(III.5) - hyp.4 + hyp.7 + (III.3) + (III.4)

In the following, we assume that $\sqcup \overrightarrow{\sigma}^{0}_{pc} \leq \sigma$ (hyp.7) and we proceed by case analysis on the possible transitions of $\mathbf{NI}^{step}_{monitor}$ for execution 0. There are three cases three consider:

1. The extended monitor performs a transition corresponding to the original monitor (Rule [Standard Execution]) because $s_0 \notin \mathit{Intercept}$;
2. The extended monitor performs a transition corresponding to the original monitor (Rule [Standard Execution]) because $\langle \mu_0, \mathsf{SubExpressions}[[s]]\rangle \notin \mathit{dom}(\mathcal{R}_{API})$;
3. The extended monitor performs a transition corresponding to the invocation of an API (Rule [API Execution]).

[Standard Execution - 1]. $s_0 \in Intercept$ (hyp.8). It follows that:

- $\sqcup \overrightarrow{\sigma}^1_{pc} \leq \sigma$ (1) - hyp.4 + hyp.7
- $s_0 \equiv s_1$ (2) - hyp.4 + hyp.8
- $s_1 \notin Intercept$ (3) - *Correctness of the Interceptor*
- $\langle \mu'_0, s'_0, \overrightarrow{\sigma}^{0'}_{pc}, \Gamma'_0, \Sigma'_0, \overrightarrow{\sigma}_{0'} \rangle \approx_{\beta',\sigma} \langle \mu'_1, s'_1, \overrightarrow{\sigma}^{1'}_{pc}, \Gamma'_1, \Sigma'_1, \overrightarrow{\sigma}_{1'} \rangle$ for some β' extending β (4) - hyp.1 + hyp.3 - hyp.6 + hyp.8 + (3)

[Standard Execution - 2]. $s_0 \in Intercept$ (hyp.8) and $\langle \mu_0, \mathsf{SubExpressions}[[s_0]] \rangle \notin dom(\mathcal{R}_{API})$ (hyp.9). It follows that:

- $\sqcup \overrightarrow{\sigma}^1_{pc} \leq \sigma$ (1) - hyp.4 + hyp.7
- $s_0 \equiv s_1$ (2) - hyp.4 + hyp.7
- $s_1 \in Intercept$ (3) - *Correctness of the Interceptor*
- $\mathsf{SubExpressions}[[s_0]], \lfloor \overrightarrow{\sigma}_0 \rfloor_n \approx_{\beta,\sigma} \mathsf{SubExpressions}[[s_1]], \lfloor \overrightarrow{\sigma}_1 \rfloor_n$, for some integer n (4) - hyp.4 + hyp.7
- $\langle \mu_1, \mathsf{SubExpressions}[[s_1]] \rangle \notin dom(\mathcal{R}_{API})$ (5) - hyp.2 + hyp.9 + (4)
- $\langle \mu'_0, s'_0, \overrightarrow{\sigma}^{0'}_{pc}, \Gamma'_0, \Sigma'_0, \overrightarrow{\sigma}_{0'} \rangle \approx_{\beta',\sigma} \langle \mu'_1, s'_1, \overrightarrow{\sigma}^{1'}_{pc}, \Gamma'_1, \Sigma'_1, \overrightarrow{\sigma}_{1'} \rangle$, for some β' extending β (6) - hyp.1 + hyp.3 - hyp.8 + (5)

[API Execution]. $s_0 \in Intercept$ (hyp.8) and $\langle \mu_0, \mathsf{SubExpressions}[[s_0]] \rangle \in dom(\mathcal{R}_{API})$ (hyp.9). It follows that:

- $\sqcup \overrightarrow{\sigma}^1_{pc} \leq \sigma$ (1) - hyp.4 + hyp.7
- $s_0 \equiv s_1$ (2) - hyp.4 + hyp.7
- $s_1 \in Intercept$ (3) - *Correctness of the Interceptor*
- $\mathsf{SubExpressions}[[s_0]], \lfloor \overrightarrow{\sigma}_0 \rfloor_n \approx_{\beta,\sigma} \mathsf{SubExpressions}[[s_1]], \lfloor \overrightarrow{\sigma}_1 \rfloor_n$, for some integer n (4) - hyp.4 + hyp.7
- $\langle \mu_1, \mathsf{SubExpressions}[[s_1]] \rangle \in dom(\mathcal{R}_{API})$ (5) - hyp.2 + hyp.9 + (4)
- $\langle \mu'_0, s'_0, \overrightarrow{\sigma}^{0'}_{pc}, \Gamma'_0, \Sigma'_0, \overrightarrow{\sigma}_{0'} \rangle \approx_{\beta',\sigma} \langle \mu'_1, s'_1, \overrightarrow{\sigma}^{1'}_{pc}, \Gamma'_1, \Sigma'_1, \overrightarrow{\sigma}_{1'} \rangle$, for some β' extending β (6) - hyp.2 + hyp.3 - hyp.8 + (5)

4 A Meta-Compiler for Securing Web APIs

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 [18]), or one inlines the monitor in the original program (as in [21] and [11]). In this section, we propose a way of extending an information flow monitor-inlining compiler in order to take into account the execution of arbitrary APIs. To this end, we assume the existence of two inlining compilers specified by two functions $\mathcal{C}_e$ and $\mathcal{C}_s$ for compiling expressions and statements respectively. The compilation function $\mathcal{C}_s$ makes use of the compilation function $\mathcal{C}_e$. The compiler $\mathcal{C}_e$ ($\mathcal{C}_s$ resp.) maps every JavaScript expression e (statement s resp.) to a pair $\langle s', i \rangle$, where:

1. s' is the statement that simulates the execution of e (s resp.) in the monitored semantics;

2. i is an index such that, after the execution of s', (1) the compiler variable $\$\hat{v}_i$ stores the value to which e (s resp.) 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 [11, 21], 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 $\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$struct \mapsto L]$.

In order to introduce the notion of *correct compiler*, one must start by defining a *similarity relation* 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 according to a new version of the β-equality called $\mathcal{C}(\beta)$-equality. This relation, denoted by $\sim_{\mathcal{C}(\beta)}$, differs from $\sim_\beta$ in that it relates each parsed function with its corresponding compilation and in that it allows the domain of an instrumented object to be larger than the one of its original counterpart.

Definition 6 (Memory Similarity). *A memory μ labeled by $\langle \Gamma, \Sigma \rangle$ is similar to a memory μ' w.r.t. β, written $\langle \mu, \Gamma, \Sigma \rangle \; \mathcal{S}_\beta \; \mu'$, if and only if $dom(\beta) = dom(\mu)$ and for every reference $r \in dom(\beta)$, if $o = \mu(r)$ and $o' = \mu'(\beta(r))$, then $\Sigma(r) = o'(\$struct)$ and for all properties $p \in dom(o)$, $o(p) \sim_{\mathcal{C}(\beta)} o'(p)$ and $\Gamma(r)(p) = o'(\$l_p)$.*

We say that an inlining compiler is *correct* w.r.t. to a given monitored semantics $\rightarrow_{IF}$ if, whenever 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 similar.

Definition 7 (Compiler Correctness). *An inlining compiler $\mathcal{C}$ is said to be correct if and only if for any two configurations $\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \epsilon \rangle$ and $\langle \mu', s' \rangle$ and function β, such that $\langle \mu, \Gamma, \Sigma \rangle \; \mathcal{S}_\beta \; \mu'$ and $\mathcal{C}\langle s \rangle = \langle s', i \rangle$, for some index i; there exists $\langle \mu_f, v_f, \overrightarrow{\sigma}_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$ such that $\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_f, v_f, \sigma_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$ iff there exists $\langle \mu'_f, v'_f \rangle$ such that $\langle \mu', s' \rangle \rightarrow^* \langle \mu'_f, v'_f \rangle$, in which case: (1) $\langle \mu_f, \Gamma_f, \Sigma_f \rangle \; \mathcal{S}_{\beta'} \; \mu'_f$, (2) $v_f \sim_{\mathcal{C}(\beta')} v'_f$, and (3) $\langle \mu'_f, \$\hat{l}_i \rangle \; \mathcal{R}_{Scope} \; \sigma_f$.*

4.1 IFlow Signatures

In order to simulate the monitored execution of API relations, we propose that each API relation be associated with three special methods – *domain*, *check*, and *label* – that we call its *IFlow Signature*. Concretely, *domain* checks whether or not to apply the API, *check* checks whether 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: $\#check$, $\#label$, and $\#domain$ are the references of the function objects corresponding to the *check*, *label*, and *domain* functions, respectively.

We require the existence of a runtime function that simulates the API Register, which we denote by $\$apiRegister$. The function $\$apiRegister$ makes use of the methods *domain* 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*. Figure 2 presents a new meta-compiler, $\mathcal{C}_{API}$, that receives as input an inlining compiler for JavaScript expressions, $\mathcal{C}_e$, 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 only expression redexes can be used as intercepting points, the compilation function for statements is left unchanged except that instead of using the original compilation function for expressions, it uses its image by the meta-compiler. The specification of the meta-compiler makes use of a syntactic function Replace that receives as input an expression and a sequence of values and outputs the expression that results from replacing in the original expression each one of its subexpressions with the corresponding sequence value. The set of expressions Intercept is assumed to contain all expressions that can reduce to an expression redex in $Intercept$. Each expression that can be an intercepting 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 whether the sequence of values corresponding to the subexpressions of the expression to compile is associated with an IFlow signature, (3 - *true*) if it is the case, 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, (3 - $false$) if it is not, it executes the compilation of an expression equivalent to the compilation of 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.

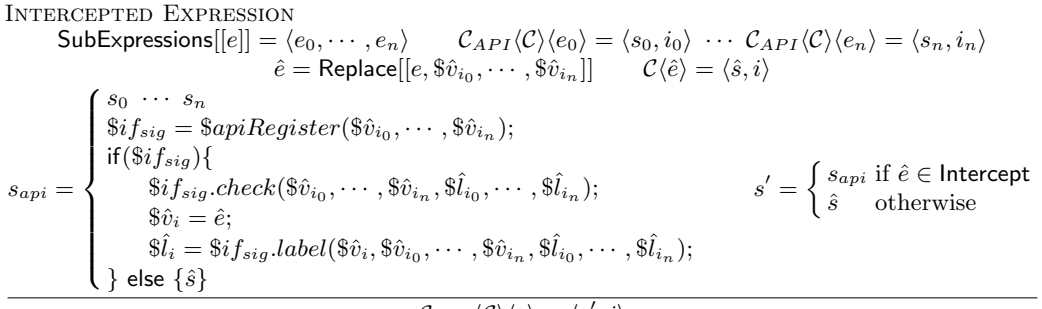

Fig. 2. Extended Compiler - $\mathcal{C}_{API}$

The correcteness of the compiler generated by the meta-compiler depends on the correctness of the compiler given as input and the correcness of the IFlow signatures in the runtime API register. Definitions 8 and 9 formally specify the conditions that the instrumented API register must verify in order for the generated compiler to be correct. Theorem 2 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.

Definition 8 (Correct IFlow Signature). *An IFlow Signature $\langle \#c, \#l, \#d \rangle$ is correct w.r.t. an API $\Downarrow_{API}$ if for any two 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, $\langle \mu_0, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \ \Downarrow_{API} \ \langle \mu_0', \Gamma', \Sigma', v_0, \sigma \rangle$ if and only if (1) $\langle \mu_1, \#c(\beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{JS} \langle \mu_1', true \rangle$, (2) $\langle \mu_1', \beta(\overrightarrow{v}) \rangle \Downarrow^{JS}_{API} \langle \mu_1'', v_1 \rangle$, and (3) $\langle \mu_1'', \#l(v_1, \beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{JS} \langle \mu_1''', \sigma \rangle$, in which case $\langle \mu_0', \Gamma', \Sigma' \rangle \ \mathcal{S}_{\beta'} \ \mu_1'''$ and $v_0 \sim_{\mathcal{C}(\beta')} v_1$, for some β' extending β.*

Definition 9 (Correct Runtime API Register). *A runtime API register corresponding to a function object pointed by #$apiRegister is correct w.r.t. an API register $\mathcal{R}_{API}$ if for any two memories μ_0 and μ_1, labeling $\langle \Gamma, \Sigma \rangle$, sequence of values $\overrightarrow{v}$, such that $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu_1$ for some function β, $\mathcal{R}_{API}(\mu_0, \overrightarrow{v}) = \ \Downarrow_{API}$ if and only if (1) $\langle \mu_1, \#apiRegister(\beta(\overrightarrow{v})) \rangle \rightarrow^*_{JS} \langle \mu_1', r_{sig} \rangle$, (2) $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_{\beta'} \ \mu_1'$ for some β' extending β, and (3) $\langle o_{sig}(\text{"check"}), o_{sig}(\text{"label"}), o_{sig}(\text{"domain"}) \rangle$ is a correct signature w.r.t. $\Downarrow_{API}$, where $o_{sig} = \mu_1'(r_{sig})$.*

Theorem 2 (Correctness). *Let $\mathcal{C}$ be correct w.r.t. $\rightarrow_{IF}$, then $\mathcal{C}_{API}\langle \mathcal{C} \rangle$ is correct w.r.t. $\mathcal{E}(\rightarrow_{IF}, Intercept, \mathcal{R}_{API})$ provided that the runtime API register is correct w.r.t. $\mathcal{R}_{API}$.*

Proof. In order to prove the claim of the theorem, we need to prove both sides of the equivalence. Since they are analogous, we give the proof of the right-to-left implication. Concretely, given a μ labeled by $\langle \Gamma, \Sigma \rangle$ and an instrumented memory μ' such that:

- $\langle \mu, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu'$ (hyp.1)
- $\mathcal{C}_{API}\langle \mathcal{C} \rangle e = \langle s, i \rangle$ (hyp.2)
- $\langle \mu', \$pc \rangle \ \mathcal{R}_{Scope} \ \sqcup \overrightarrow{\sigma}_{pc} = \sigma_{pc}$ (hyp.3)
- $\langle \mu, s, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_f, v_f, \sigma_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$ (hyp.4)

we have to prove that:

- $\langle \mu, s \rangle \rightarrow^*_{JS} \langle \mu'_f, v'_f \rangle$
- $\langle \mu_f, \Gamma_f, \Sigma_f \rangle \ \mathcal{S}_\beta \ \mu'_f$

We proceed by induction on the number of API calls occuring in e - n.

[N = 0]. Let $n = 0$ (hyp.5) and $\mathsf{SubExpressions}[[e]] = \langle e_1, \cdots, e_k \rangle$ (hyp.6). We conclude:

- $\langle \mu, e_1, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_1, v_1, \sigma_{pc}, \Gamma_1, \Sigma_1, \sigma_1 \rangle$, ...,
  $\langle \mu_{k-1}, e_k, \overrightarrow{\sigma}_{pc}, \Gamma_{k-1}, \Sigma_{k-1}, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_k, v_k, \sigma_{pc}, \Gamma_k, \Sigma_k, \sigma_k \rangle$ (1) - hyp.4 + hyp.6
- $\langle \mu', e_1 \rangle \rightarrow^*_{JS} \langle \mu'_1, v'_1 \rangle$, ..., $\langle \mu'_{k-1}, e_k \rangle \rightarrow^*_{JS} \langle \mu'_k, v'_k \rangle$
  (2) - *Correctness* of $\mathcal{C}$ + hyp.1-hyp.3
- $\langle \mu_k, \Gamma_k, \Sigma_k \rangle \ \mathcal{S}_{\beta_k} \ \mu'_k$, for some β_k extending β (3) - *Correctness* of $\mathcal{C}$ + hyp.1-hyp.3
- $\langle \mu'_k, \#apiRegister() \rangle \rightarrow^*_{JS} \langle \mu''_k, null \rangle$ (4) - *Correctenss* of the API register
- $\langle \mu_k, \Gamma_k, \Sigma_k \rangle \ \mathcal{S}_{\beta_k} \ \mu''_k$ (5) - *Correctenss* of the API register
- $\langle \mu_k, \mathsf{Replace}[[e, v_1, \cdots, v_k]], \sigma_{pc}, \Gamma_k, \Sigma_k, \langle \sigma_1, \cdots, \sigma_k \rangle \rangle \rightarrow_{IF} \langle \mu_f, v_f, \sigma_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$
  (6) - hyp.4 + hyp.6
- $\langle \mu'_k, \mathsf{Replace}[[e, v'_1, \cdots, v'_k]] \rangle \rightarrow_{JS} \langle \mu'_f, v'_f \rangle$ (7) - *Correctness* of $\mathcal{C}$ + (5) + (6)
- $\langle \mu_f, \Gamma_f, \Sigma_f \rangle \ \mathcal{S}_{\beta_f} \ \mu'_f$, for some β_f (8) - *Correctenss* of the API register

[N = L+1]. We proceed by induction on the number of nested subexpressions - j. Suppose that $j = 0$ (hyp.5). Since there are no nested subexpressions and the number of API calls is greater than 0 it follows that the current expression triggers an API call. We conclude that:

- $n = 1$ (1) - hyp.5 + $n > 0$
- $\langle \mu, e, \sigma_{pc}, \Gamma, \Sigma, \epsilon \rangle \rightarrow_{IF} \langle \mu_f, v_f, \sigma_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$ (2) - hyp.4 + hyp.5 + $n > 0$
- $\langle \mu', \#apiRegister() \rangle \rightarrow^*_{JS} \langle \mu'', r_{sig} \rangle$ (3) - *Correctenss* of the API register
- $\langle \mu'', \#c() \rangle \rightarrow^*_{JS} \langle \mu'_c, true \rangle$
  $\langle \mu'_c, \epsilon \rangle \Downarrow^{JS}_{API} \langle \mu'_a, v'_f \rangle$
  $\langle \mu'_a, \#l() \rangle \rightarrow^*_{JS} \langle \mu'_f, \sigma \rangle$ (4) - *Correctness for IFlow Signatures*
- $\langle \mu_f, \Gamma_f, \Sigma_f \rangle \ \mathcal{S}_{\beta_f} \ \mu'_f$, for some β_f (8) - *Correctenss* of the IFlow Signature

Suppose $j = i + 1$ (hyp.5). There are two cases to consider: either e is intercepted or e is not intercepted. Suppose that e is intercepted (hyp.6). Suppose $\mathsf{SubExpressions}[[e]] = \langle e_1, \cdots, e_k \rangle$ (hyp.7). We conclude:

- $\langle \mu, e_1, \overrightarrow{\sigma}_{pc}, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_1, v_1, \sigma_{pc}, \Gamma_1, \Sigma_1, \sigma_1 \rangle$, ...,
  $\langle \mu_{k-1}, e_k, \overrightarrow{\sigma}_{pc}, \Gamma_{k-1}, \Sigma_{k-1}, \epsilon \rangle \rightarrow^*_{IF} \langle \mu_k, v_k, \sigma_{pc}, \Gamma_k, \Sigma_k, \sigma_k \rangle$ (1) - hyp.4 + hyp.6
- $\langle \mu', e_1 \rangle \rightarrow^*_{JS} \langle \mu'_1, v'_1 \rangle$, ..., $\langle \mu'_{k-1}, e_k \rangle \rightarrow^*_{JS} \langle \mu'_k, v'_k \rangle$
  (2) - hyp.1-hyp.3 + **ih** (external induction hypothesis)
- $\langle \mu_k, \Gamma_k, \Sigma_k \rangle \ \mathcal{S}_{\beta_k} \ \mu'_k$, for some β_k extending β (3) - *Correctness* of $\mathcal{C}$ + hyp.1-hyp.3
- $\langle \mu_k, \Gamma_k, \Sigma_k \rangle \ \mathcal{S}_{\beta_k} \ \mu'_k$, for some β_k extending β (3) - *Correctness* of $\mathcal{C}$ + hyp.1-hyp.3
- $\langle \mu'_k, \#apiRegister() \rangle \rightarrow^*_{JS} \langle \mu''_k, null \rangle$ (4) - *Correctenss* of the API register
- $\langle \mu_k, \Gamma_k, \Sigma_k \rangle \ \mathcal{S}_{\beta_k} \ \mu''_k$ (5) - *Correctenss* of the API register
- $\langle \mu_k, \mathsf{Replace}[[e, v_1, \cdots, v_k]], \sigma_{pc}, \Gamma_k, \Sigma_k, \langle \sigma_1, \cdots, \sigma_k \rangle \rangle \rightarrow_{IF} \langle \mu_f, v_f, \sigma_{pc}, \Gamma_f, \Sigma_f, \sigma_f \rangle$
  (6) - hyp.4 + hyp.6

– $\langle \mu'_k, \mathsf{Replace}[[e, v'_1, \cdots, v'_k]] \rangle \rightarrow_{JS} \langle \mu'_f, v'_f \rangle$ (7) - *Correctness* of $\mathcal{C}$ + (5) + (6)
– $\langle \mu_f, \Gamma_f, \Sigma_f \rangle \ \mathcal{S}_{\beta_f} \ \mu'_f$, for some β_f (8) - *Correctenss* of the API register

An implementation of the presented meta-compiler as well as a proven correct inlining compiler can be found in [1] together with an online testing tool and a set of IFlow signatures that includes all those studied in the paper.

5 Securing Information Flow in the DOM API

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [23]. In order to access the functionalities of this API, JavaScript programs manipulate a special kind of objects, here named *DOM objects*. There are several different kinds of DOM objects, for instance objects to represent document elements, of type *Element*, and objects to represent the attributes of document elements, of type *Attribute*. We partially model DOM Element objects, whose behavior is established in the DOM Core Level 1 API [23].

In contrast to the ECMA Standard [2] that specifies in full detail the internals of objects created during the execution, the DOM API only specifies the behavior that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine. 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.

In modeling DOM Element objects, we do not include properties that can be computed from properties that are already modeled, such as the length property. Interestingly, by defining the property look-up as an extension point of the semantics, we can easily model the behavior associated with the looking-up of these extra properties. Concretely, each one of these properties is associated with an API relation that is executed each time a program looks-up the corresponding value in a DOM Element object. Hence, every time we want to extend the model with a property that can be computed from the properties that are explicitly modeled, we do not need to modify every rule that manipulates the DOM objects that are supposed to define this property (and the corresponding proofs of noninterference). Instead, we only need to specify the API relation corresponding to the looking-up of that property, include it in the API register ($\mathcal{R}_{API}$), and prove it to be noninterferent. This approach to modeling the properties of DOM objects has the clear advantage of allowing us to modularly reason about the corresponding information leaks.

In the fragment of the DOM Core Level 1 API modeled in this work, we depart from the specification in that in our model the child nodes of a given DOM Element node are directly accessed through their parent instead of through a special object *childNodes*. Hence, instead of writing $div1.childNodes[i]$ to ac-

cess the ith child of the DIV element bound to $div1$, we simply write $div1[i]$. Furthermore, we say that, provided that it is defined, i is an index of $div1$.

The special features of the DOM API spawn new implicit information flows as the following examples illustrate.

Example 10 (Leak via removeChild - Order Leak). Suppose that in the original memory there are two orphan DIV nodes bound to variables $div1$ and $div2$.

```
1  div1.appendChild(div2);
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 l can be either $div2$ or *undefined*, meaning that the final level associated with variable l must be H in both executions.

Example 11 (Leak via appendChild - Order Leak). Suppose that in the original memory there are three orphan DIV nodes with low structure security level bound to variables $div1$, $div2$, and $div3$.

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

After the execution of this program, depending on the value of the high variable h, l can be either set to a reference pointing to $div3$ or to a reference pointing to $div2$. Hence, for this example to be legal, the reading effect of $div1[0]$ must be H no matter the value of the non-observable variable h. Notice that this kind of leak cannot be directly expressed in the model of [25].

The examples above show that, when appending a new node to a given Element 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 model DOM objects as standard JavaScript objects. Furthermore, we assume that every memory contains a *document* object denoted *doc*, which is accessed through the property "doc" of the global object and which is assumed to be stored in a fixed reference $\#doc$. Each DOM object is assumed to define 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 that point to each of its $n+1$ children. The DOM Element objects in memory form a *forest*, that is, a set of *trees* of Element objects, such that the displayed HTML document corresponds to the tree hanging from the object pointed to by $\#doc$.

Figure 3 presents the labeled API relations with which we extend the JavaScript semantics for interaction with DOM objects. In the specification of these API relations we make use of the following four semantic functions:

- $\mathcal{R}_{\#Children}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathbb{N}$, such that if $\langle r, n\rangle \in \mathcal{R}_{\#Children}(\mu)$, then the DOM node pointed to by r has n children (meaning that it defines the indexes $0, \cdots, n-1$).
- $\mathcal{R}_{Ancestor}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{Ancestor}(\mu)$, 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.
- $\mathcal{R}_{Parent}$ receives a memory μ as input and outputs a relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{Parent}(\mu)$, 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.

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 _. The monitored API relations given in Figure 3 enforce the invariant on the indexes of every DOM Element object discussed above. *Indexes Invariant:* for any memory μ well-labeled by $\langle \Gamma, \Sigma\rangle$, reference $r \in dom(\mu)$ pointing to a DOM Element object, and two indexes $i, j \in dom(\Gamma(r))$ such that $i < j$, the following hold: $\Gamma(r)(i) \leq \Gamma(r)(j) \leq \Sigma(r)$.

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 functions as the level of a "ghost" property pointing to its parent node. Hence, we strenghten the low-equality relation for DOM objects in the following way: if $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$, $\beta(r_0) = r_1$, $\langle r_0^p, r_0\rangle \in \mathcal{R}_{Parent}(\mu_0)$, and $\langle r_1^p, r_1\rangle \in \mathcal{R}_{Parent}(\mu_1)$, either $(\Sigma_0(r) \sqcup \Sigma_1(\beta(r)) \leq \sigma \wedge r_0^p \sim_\beta r_1^p)$ or $\Sigma_0(r) \sqcap \Sigma_1(\beta(r)) \not\leq \sigma$.

In the following, we give a brief explanation for each of the rules in Figure 3. [createElement] The API relation $\Downarrow_{cre}$ 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 4). [removeChild] The API relation $\Downarrow_{rem}$ 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 10). [appendChild] The API relation $\Downarrow_{app}$ 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 $\Downarrow_{app}$ 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

CREATEELEMENT

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

REMOVECHILD

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

APPENDCHILD - ORPHAN NODE

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

APPENDCHILD - NON-ORPHAN NODE

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

LENGTH

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

PARENTNODE

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

INDEX

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

NEXTSIBLING - NON-ORPHAN NODE

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

NEXTSIBLING - ORPHAN NODE

$$\frac{r \in \mathcal{O}rphan(\mu)}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{sib}\; \langle \mu, \Gamma, \Sigma, \textit{undefined}, \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r)\rangle}$$

Fig. 3. DOM APIs - Building and Traversing DOM Trees

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 $\Downarrow_{app}$ 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 $\Downarrow_{len}$ 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 $\Downarrow_{par}$ 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 corrresponding parent node. [INDEX] The API relation $\Downarrow_{ind}$ 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 $\mu(r)$. [NEXTSIBLING] The API relation $\Downarrow_{sib}$ 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 $\mu(r)$. In order for these API relations to be added to the semantics, one has to add them to the the API register. Hence, we assume that the $\mathcal{R}_{API}$ extends the API register given in Figure 4.

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

Fig. 4. $\mathcal{R}^{DOM}_{API}$

Lemma 18 validates the hypothesis of Theorem 1 for $\mathcal{R}^{DOM}_{API}$, meaning that the extension of the security monitor with the APIs in the range of $\mathcal{R}^{DOM}_{API}$ does not entail a violation of the security theorem.

Lemma 1 (Noninterference for the DOM API). NI($\mathcal{R}^{DOM}_{API}$).

Proof. Applying Lemmas 3, 6, 9, 11, 13, 15, and 17, we conclude that all API relations in the range of $\mathcal{R}_{API}^{DOM}$ are noninterferent. It remains to prove that given two memories μ and μ', two labelings $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$, a sequence of values $\overrightarrow{v}$, a security level σ, and a function β, such that $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ (hyp.1), then $\mathcal{R}_{API}^{DOM}(\mu, \overrightarrow{v}) = \mathcal{R}_{API}^{DOM}(\mu', \beta(\overrightarrow{v}))$. We proceed by case analysis:

- $\mathcal{R}_{API}^{DOM}(\mu, \overrightarrow{v}) = \Downarrow_{cre}$ (hyp.2). We conclude that: $\overrightarrow{v} = \langle \#doc, \text{“createElement”} \rangle$, implying that $\beta(\overrightarrow{v}) = \langle \#doc, \text{“createElement”} \rangle$ (because β is proper), and therefore that: $\mathcal{R}_{API}^{DOM}(\mu', \beta(\overrightarrow{v})) = \Downarrow_{cre}$.
- $\mathcal{R}_{API}^{DOM}(\mu, \overrightarrow{v}) = \Downarrow_{app}$ (hyp.2). We conclude that: $\overrightarrow{v} = \langle r, \text{“appendChild”} \rangle$ and $@tag \in dom(\mu(r))$, implying that: $\beta(\overrightarrow{v}) = \langle \beta(r), \text{“createElement”} \rangle$, $@tag \in dom(\mu'(r'))$ (*Tag Invariant*), and therefore that: $\mathcal{R}_{API}^{DOM}(\mu', \beta(\overrightarrow{v})) = \Downarrow_{app}$. All the remaining cases are equivalent and therefore ommitted.

Figure 5 presents a possible IFlow signature for the API relation $\Downarrow_{rem}$, which makes use of the two following runtime functions: (1) $\$check$ diverges if its argument is different from *true* and returns *true* otherwise and (2) $\$shadow$ receives as input a property name and outputs the name of the corresponding shadow property.

```
check = function(o0, m, o2, σ0, σ1, σ2){
    var i = 0;
    while(i ≤ o0["length"]){
        if(o0[i] == o2) break;
    }
    $check(i < o0["length"]);
    o2[$index] = i;
    return $check(σ0 ⊔ σ1 ⊔ σ2 ≤ o0[$shadow(i)]);
}

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

```
label = function(ret, o0, m, o2, σ0, σ1, σ2){
    var i = o2[$index];
    var j = o0["length"];
    while(j > i){
        o0[$shadow(j)] = o0[$shadow(j + 1)];
    }
    delete o0[$shadow(o0["length"])];
    delete o2[$index];
    return σ0 ⊔ σ1 ⊔ σ2;
}
```

Fig. 5. IFlow Signature of $\Downarrow_{rem}$

6 Conclusion

In summary, we have proposed a methodology for extending arbitrary monitored semantics with secure APIs, which allows us to prove the security of the extended monitor in a modular way. As a case study, we applied the methodology to an important fragment of the DOM Core Level 1 API for which we have studied information leaks not explored in previous related work. The proofs of the results as well as an implementation that includes the IFlow signatures of the APIs studied in the paper can be found in [1].

References

1. Information flow monitor-inlining compiler. http://www-sop.inria.fr/indes/ifJS.
2. The 5.1th edition of ECMA 262 June 2011. ECMAScript Language Specification. Technical report, ECMA, 2011.
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. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.
8. 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. To appear.
9. A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS*, 2012.
10. A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, 2009.
11. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *CSF*, 2010.
12. D. Crockford. Adsafe. http://www.adsafe.org.
13. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.
14. The FaceBook Team: FBJS. http://wiki.developers.facebook.com/index.php/FBJS.
15. P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. Dom: Towards a formal specification. In *PLAN-X*, 2008.
16. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
17. A. Guha, B. Lerner, J. Gibbs Politz, and S. Krishnamurthi. Web api verification: Results and challenges. 2012.
18. D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*, 2012.
19. S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In *APLAS*, 2008.
20. S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *CSF*, 2009.
21. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 2012.
22. J. Gibbs Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: Type-based verification of javascript sandboxing. In *USENIX Security Symposium*, 2011.
23. W3C Recommendation. DOM: Document Object Model (DOM). Technical report, W3C, 2005.
24. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.

25. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
26. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
27. P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, 2007.
28. A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *SP*, 2011.
29. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.
30. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS*, 2006.

A Proofs of Section 5

Lemma 2 (Confinement for $\Downarrow_{cre}$). *$\Downarrow_{cre}$ is confined.*

Proof. Observing that the reference of the newly created object is not in the domain of μ, the result follows immediately.

Lemma 3 (Noninterference for $\Downarrow_{cre}$). $\mathbf{NI}(\Downarrow_{cre})$.

Proof. Hypotheses:

- $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
- $\langle \_, \_, m\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle \approx_{\beta,\sigma} \langle \_, \_, m'\rangle, \langle \sigma'_0, \sigma'_1, \sigma'_2\rangle$ hyp.2
- $\langle \mu, \Gamma, \Sigma, \langle \_, \_, m\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{cre} \langle \mu_f, \Gamma_f, \Sigma_f, r, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle$ hyp.3
- $\langle \mu', \Gamma', \Sigma', \langle \_, \_, m'\rangle, \langle \sigma'_0, \sigma'_1, \sigma'_2\rangle\rangle \Downarrow_{cre} \langle \mu'_f, \Gamma'_f, r', \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2\rangle$ hyp.4

We conclude that:

- $r \notin dom(\mu)$ (1) - hyp.3
- $\mu_f = \mu\,[r \mapsto [@tag \mapsto m]]$ (2) - hyp.3
- $\Gamma_f = \Gamma\,[r \mapsto [@tag \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]]$ (3) - hyp.3
- $\Sigma_f = \Sigma\,[r \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]$ (4) - hyp.3
- $r' \notin dom(\mu')$ (5) - hyp.4
- $\mu'_f = \mu'\,[r' \mapsto [@tag \mapsto m']]$ (6) - hyp.4
- $\Gamma'_f = \Gamma'\,[r' \mapsto [@tag \mapsto \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2]]$ (7) - hyp.4
- $\Sigma'_f = \Sigma'\,[r' \mapsto \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2]$ (8) - hyp.4

We consider two different cases: (I) $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \sigma$ and (II) $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\leq \sigma$.

Case: $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \sigma$ (hyp.5) and $\beta' = \beta\,[r \mapsto r']$ (hyp.6):

- $m = m'$ (9) - hyp.2 + hyp.5
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \leq \sigma$ (10) - hyp.2 + hyp.5
- $\mathcal{P}arent(\mu_f, r) = \mathcal{P}arent(\mu'_f, r') = \mathit{undefined}$ (11) - (1) + (2) + (5) + (6)
- $\Big(\Sigma_f(r) \leq \sigma \vee \Sigma'_f(r') \leq \sigma\Big) \Rightarrow \begin{cases} \mathcal{P}arent(\mu_f, r) \sim_{\beta'} \mathcal{P}arent(\mu'_f, r') \\ \Sigma_f(r) \sqcup \Sigma'_f(r') \leq \sigma \end{cases}$ (12) - hyp.5 + (4) + (8) + (9)

- $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta',\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$ (13) - hyp.1 + hyp.6 + (1) - (8) + (10) + (12)
- $r, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \approx_{\beta,\sigma} r', \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2$ (14) - hyp.2 + hyp.6

Case: $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\leq \sigma$ (hyp.5):

- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \not\leq \sigma$ (9) - hyp.2 + hyp.5
- $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu_f, \Gamma_f, \Sigma_f$ (10) - hyp.3 + hyp.5 + Confinement for $\Downarrow_{cre}$ (Lemma 2)
- $\mu', \Gamma', \Sigma' \approx_{id,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$ (11) - hyp.4 + (9) + Confinement for $\Downarrow_{cre}$ (Lemma 2)
- $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$ (12) - hyp.1 + (10) + (11) + Lateral Transitivity of $\approx_{\beta,\sigma}$
- $r, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \approx_{\beta',\sigma} r', \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2$ (13) - hyp.5 + (9)

Lemma 4 (Strong Cofinement for $\Downarrow_{rem}$). *Let μ be a memory, $\langle \Gamma, \Sigma \rangle$ a labeling, r_0 and r_2 two references, $\sigma_0, \sigma_1, \sigma_2, \sigma \in \mathcal{L}$ four security levels , such that: (1) $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2 \rangle, \langle \sigma_0, \sigma_1, \sigma_2 \rangle \rangle \Downarrow_{rem} \langle \mu', \Gamma', \Sigma', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \rangle$ for some memory μ' labeled by $\langle \Gamma', \Sigma' \rangle$, (2) $\mu(r_0)(i) = r_2$ for some index i, and (3) $\Gamma(r_0)(i) \not\leq \sigma$; then: $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$.*

Proof. We conclude that there an integer n, an object o_0, and a labeling object γ_0, such that:

- $\langle r_0, n+1 \rangle \in \mathcal{R}_{\#Children}(\mu)$ (1) - hyp.1 + hyp.2
- $dom(o_0) = dom(\gamma_0) = dom(\mu(r_0)) \backslash \{n\}$ (2) - hyp.1 + hyp.2
- $\forall_{0 \leq j < i} \cdot o_0(j) = \mu(r_0)(j)$, $\forall_{i \leq j < n} \cdot o_0(j) = \mu(r_0)(j+1)$, $o_0(@tag) = \mu(r_0)(@tag)$ (3) - hyp.1 + hyp.2
- $\forall_{0 \leq j < i} \cdot \gamma_0(j) = \Gamma(r_0)(j)$, $\forall_{i \leq j < n} \cdot \gamma_0(j) = \Gamma(r_0)(j+1)$, $\gamma_0(@tag) = \Gamma(r_0)(@tag)$ (4) - hyp.1 + hyp.2
- $\mu' = \mu\,[r_0 \mapsto o_0]$ and $\Gamma' = \Gamma\,[r_0 \mapsto \gamma_0]$ (5) - hyp.1 + hyp.2
- $\forall_{i \leq j \leq n} \cdot \Gamma(r_0)(j) \not\leq \sigma$ and $\Sigma(r_0) \not\leq \sigma$ (6) - hyp.2 + (1) + *Indexes Invariant*
- $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$ (7) - (2) - (5) + (6) + High Object Update Lemma

Lemma 5 (Cofinement for $\Downarrow_{rem}$). $\Downarrow_{\mathsf{rem}}$ *is confined.*

Proof. Hypotheses:

- $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2 \rangle, \langle \sigma_0, \sigma_1, \sigma_2 \rangle \rangle \Downarrow_{rem} \langle \mu', \Gamma', \Sigma', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \rangle$ hyp.1
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\leq \sigma$ hyp.2

From Hypothesis hyp.1, we conclude that there is an index i such that $\mu(r_0)(i) = r_2$. The semantics of $\Downarrow_{rem}$ impose that: $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Gamma(r_0)(i)$. Therefore, we conclude that: $\Gamma(r_0)(i) \not\leq \sigma$. Applying the Strong Confinement Lemma for $\Downarrow_{rem}$ (Lemma 4), the result immediately follows.

Lemma 6 (Noninterference for $\Downarrow_{rem}$). NI($\Downarrow_{rem}$).

Proof. Hypotheses:

- $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
- $\langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle \approx_{\beta,\sigma} \langle r'_0, \_, r'_2\rangle, \langle \sigma'_0, \sigma'_1, \sigma'_2\rangle$ hyp.2
- $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{rem} \langle \mu_f, \Gamma_f, \Sigma_f, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle$ hyp.3
- $\langle \mu', \Gamma', \Sigma', \langle r'_0, \_, r'_2\rangle, \langle \sigma'_0, \sigma'_1, \sigma'_2\rangle\rangle \Downarrow_{rem} \langle \mu'_f, \Gamma'_f, \Sigma'_f, r'_2, \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2\rangle$ hyp.4

We conclude that there are four integers i, i', n and n', two objects o_0 and o'_0, and two labeling objects γ_0 and γ'_0, such that:

- $\mu(r_0)(i) = r_2$ and $\langle r_0, n+1\rangle \in \mathcal{R}_{\#Children}(\mu)$ (1) - hyp.3
- $dom(o_0) = dom(\gamma_0) = dom(\mu(r_0))\backslash\{n\}$ (2) - hyp.3
- $\forall_{0\le j<i} \cdot o_0(j) = \mu(r_0)(j)$, $\forall_{i\le j<n} \cdot o_0(j) = \mu(r_0)(j+1)$, $o_0(@tag) = \mu(r_0)(@tag)$ (3) - hyp.3
- $\forall_{0\le j<i} \cdot \gamma_0(j) = \Gamma(r_0)(j)$, $\forall_{i\le j<n} \cdot \gamma_0(j) = \Gamma(r_0)(j+1)$, $\gamma_0(@tag) = \Gamma(r_0)(@tag)$ (4) - hyp.3
- $\mu_f = \mu\,[r_0 \mapsto o_0]$ and $\Gamma_f = \Gamma\,[r_0 \mapsto \gamma_0]$ (5) - hyp.3
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Gamma(r_0)(i)$ (6) - hyp.3
- $\mu'(r'_0)(i') = r'_2$ and $\langle r'_0, n'+1\rangle \in \mathcal{R}_{\#Children}(\mu')$ (7) - hyp.4
- $dom(o'_0) = dom(\gamma'_0) = dom(\mu'(r'_0))\backslash\{n'\}$ (8) - hyp.4
- $\forall_{0\le j<i'} \cdot o'_0(j) = \mu'(r'_0)(j)$, $\forall_{i'\le j<n'} \cdot o'_0(j) = \mu'(r'_0)(j+1)$, $o'_0(@tag) = \mu'(r'_0)(@tag)$ (9) - hyp.4
- $\forall_{0\le j<i'} \cdot \gamma'_0(j) = \Gamma'(r'_0)(j)$, $\forall_{i'\le j<n} \cdot \gamma'_0(j) = \Gamma'(r'_0)(j+1)$, $\gamma'_0(@tag) = \Gamma'(r'_0)(@tag)$ (10) - hyp.4
- $\mu'_f = \mu'\,[r'_0 \mapsto o'_0]$ and $\Gamma'_f = \Gamma'\,[r'_0 \mapsto \gamma'_0]$ (11) - hyp.4
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \le \Gamma'(r'_0)(i')$ (12) - hyp.4

We immediately conclude from the Hypothesis hyp.2 that $r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \approx_{\beta,\sigma} r'_2, \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2$. We now proceed by case analysis.

Case: $\Gamma(r_0)(i) \le \sigma$ (hyp.5) and suppose, without loss of generality that $n \le n'$ (hyp.6):

- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \sigma$ (13) - hyp.5 + (6)
- $r_0 \sim_\beta r'_0$ (14) - hyp.2 + (13)
- $r_2 \sim_\beta r'_2$ (15) - hyp.2 + (13)
- $\forall_{0\le j\le i} \cdot \Gamma(r_0)(j) \sqcup \Gamma'(r'_0)(j) \le \sigma$ (16) - hyp.1 + hyp.5 + (14) + *Indexes Invariant*
- $\forall_{0\le j\le i} \cdot \mu(r_0)(j) \sim_\beta \mu'(r'_0)(j)$ (17) - hyp.1 + hyp.5 + (14) + *Indexes Invariant*
- $r_2 \sim_\beta \mu'(r'_0)(i)$ (18) - (1) + (17)
- $\mu'(r'_0)(i) = r'_2$ (19) - (15) + (18)
- $i = i'$ (20) - (7) + (19) + *DOM Integrity*
- $\forall_{i<j\le n} \cdot \mu(r_0)(j), \Gamma(r_0)(j) \approx_{\beta,\sigma} \mu'(r'_0)(j), \Gamma'(r'_0)(j)$ (21) - hyp.1 + hyp.6 + (14)
- $\forall_{n<j\le n'} \cdot \Gamma'(r'_0)(j) \not\le \sigma$ (22) - hyp.1 + hyp.6 + (14)

– $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$
(23) - hyp.1 + (2) - (5) + (8) - (11) + (17) + (21) + (22) + + Low Object Update Lemma

Case: $\Gamma(r_0)(i) \not\leq \sigma$ (hyp.5):

– $\Gamma'(r'_0)(i') \not\leq \sigma$ (13)
Suppose $\Gamma'(r'_0)(i') \leq \sigma$ (hyp.6):
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \leq \sigma$ (13.1) - hyp.6 + (12)
- $r_0 \sim_\beta r'_0$ (13.2) - hyp.2 + (13.1)
- $r_2 \sim_\beta r'_2$ (13.3) - hyp.2 + (13.1)
- $\forall_{0 \leq j \leq i'} \cdot \Gamma(r_0)(j) \sqcup \Gamma'(r'_0)(j) \leq \sigma$
(13.4) - hyp.1 + hyp.6 + (13.2) + *Indexes Invariant*
- $\forall_{0 \leq j \leq i'} \cdot \mu(r_0)(j) \sim_\beta \mu'(r'_0)(j)$
(13.5) - hyp.1 + hyp.6 + (13.2) + *Indexes Invariant*
- $\mu(r_0)(i') \sim_\beta r'_2$ (13.6) - (7) + (13.5)
- $r_2 = \mu(r_0)(i')$ (13.7) - (13.3) + (13.6)
- $i = i'$ (13.8) - (1) + (13.7)
- $\Gamma(r_0)(i) \leq \sigma$ (13.9) - (13.4) + (13.8)
- *Contradiction* (13.10) - hyp.5 + (13.9)

– $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu_f, \Gamma_f, \Sigma_f$
(16) - hyp.3 + hyp.5 + Strong Confinement for $\Downarrow_{rem}$ (Lemma 4)
– $\mu', \Gamma', \Sigma' \approx_{id,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$
(17) - hyp.4 + (15) + Strong Confinement for $\Downarrow_{rem}$ (Lemma 4)
– $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$
(18) - hyp.1 + (16) + (17) + Lateral Transitivity of $\approx_{\beta,\sigma}$

Lemma 7 (Strong Cofinement for $\Downarrow_{app}$). *Let μ be a memory, $\langle \Gamma, \Sigma \rangle$ a labeling, r_0 and r_2 two references, and $\sigma_0, \sigma_1, \sigma_2, \sigma \in \mathcal{L}$ four security levels, such that: (1) $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2 \rangle, \langle \sigma_0, \sigma_1, \sigma_2 \rangle \rangle \Downarrow_{app} \langle \mu', \Gamma', \Sigma', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \rangle$ for some memory μ' and labeling $\langle \Gamma', \Sigma' \rangle$ and (2) $\Sigma(r_0) \sqcup \Sigma(r_2) \not\leq \sigma$; then: $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$.*

Proof. We proceed by case analysis.

Case: $r_2 \in \mathcal{O}rphan(\mu)$ (hyp.3). We conclude that there are is an integer n, such that:

– $\langle r_0, n \rangle \in \mathcal{R}_{\#Children}(\mu)$ (1) - hyp.1 + hyp.3
– $\mu' = \mu\,[r_0 \mapsto \mu(r_0)\,[n \mapsto r_2]]$ (2) - hyp.1 + hyp.3
– $\Gamma' = \Gamma\,[r_0 \mapsto \Gamma(r_0)\,[n \mapsto \Sigma(r_0)]]$ (3) - hyp.1 + hyp.3
– $\Sigma' = \Sigma$ (4) - hyp.1 + hyp.3
– $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Sigma(r_0) \sqcap \Sigma(r_2)$ (5) - hyp.1 + hyp.3
– $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma$
(6) - hyp.2 + (2) - (4) + High Object Update Lemma

Case: $r_2 \notin \mathcal{O}rphan(\mu)$ (hyp.3). We conclude that there is a reference r_p, a memory $\hat{\mu}$, a labeling $\langle \hat{\Gamma}, \hat{\Sigma} \rangle$, such that $\langle r_p, r_2 \rangle \in \mathcal{R}_{Parent}(\mu)$:

– $\langle \mu, \Gamma, \Sigma, \langle r_p, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_0), \sigma_1, \sigma_2\rangle\rangle \Downarrow_{rem} \langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \_, \_\rangle$
(1) - hyp.1 + hyp.3
– $\langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{app} \langle \mu', \Gamma', \Sigma', \_, \_\rangle$
(2) - hyp.1 + hyp.3
– $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Sigma(r_0) \sqcap \Sigma(r_2)$ (3) - hyp.1 + hyp.3
– $\mu, \Gamma, \Sigma \approx_{id,\sigma} \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}$
(4) - hyp.2 + (1) + Confinement for $\Downarrow_{rem}$ (Lemma 5)
– $\hat{\mu}, \hat{\Gamma}, \hat{\Sigma} \approx_{id,\sigma} \mu', \Gamma', \Sigma'$ (5) - hyp.2 + (2) + **Case** $r_2 \in \mathcal{O}rphan(\mu)$
– $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu', \Gamma', \Sigma'$ (6) - (4) + (5) + Transitivity of $\approx_{\beta,\sigma}$

Lemma 8 (Cofinement for $\Downarrow_{app}$). *$\Downarrow_{app}$ is confined.*

Proof. Hypotheses:

– $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{app} \langle \mu', \Gamma', \Sigma', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle$ hyp.1
– $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\leq \sigma$ hyp.2

Both rules for $\Downarrow_{app}$ impose that: $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Sigma(r_0) \sqcap \Sigma(r_2)$. Therefore, we conclude that: $\Sigma(r_0) \not\leq \sigma$. Applying the Strong Confinement Lemma for $\Downarrow_{app}$ (Lemma 7), the result immediately follows.

Lemma 9 (Noninterference for $\Downarrow_{app}$). **NI**($\Downarrow_{app}$).

Proof. Hypotheses:

– $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
– $\langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle \approx_{\beta,\sigma} \langle r_0', \_, r_2'\rangle, \langle \sigma_0', \sigma_1', \sigma_2'\rangle$ hyp.2
– $\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{app} \langle \mu_f, \Gamma_f, \Sigma_f, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle$ hyp.3
– $\langle \mu', \Gamma', \Sigma', \langle r_0', \_, r_2'\rangle, \langle \sigma_0', \sigma_1', \sigma_2'\rangle\rangle \Downarrow_{rem} \langle \mu_f', \Gamma_f', \Sigma_f', r_2', \sigma_0' \sqcup \sigma_1' \sqcup \sigma_2'\rangle$ hyp.4

From hyp.2, it immediately follows that: $r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \approx_{\beta',\sigma} r_2', \sigma_0' \sqcup \sigma_1' \sqcup \sigma_2'$, for any function β' extending β. We proceed by case analysis.

Case I: $\Sigma(r_0) \not\leq \sigma$ (hyp.5):

– $\mu, \Gamma, \Sigma \approx_{id,\sigma} \mu_f, \Gamma_f, \Sigma_f$
(1) - hyp.3 + Strong Confiment for $\Downarrow_{app}$
– $\mu', \Gamma', \Sigma' \approx_{id,\sigma} \mu_f', \Gamma_f', \Sigma_f'$
(2) - hyp.4 + Strong Confiment for $\Downarrow_{app}$
– $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta,\sigma} \mu_f', \Gamma_f', \Sigma_f'$
(3) - (1) + (2) + Lateral Transitivity of $\approx_{\beta,\sigma}$

Case II: $r_2 \notin \mathcal{O}rphan(\mu)$ (hyp.5) and $\Sigma(r_0) \leq \sigma$ (hyp.6). We conclude that there are is an integer n, such that:

– $\langle r_0, n\rangle \in \mathcal{R}_{\#Children}(\mu)$ (1) - hyp.3 + hyp.5
– $\mu_f = \mu\,[r_0 \mapsto \mu(r_0)\,[n \mapsto r_2]]$ (2) - hyp.3 + hyp.5
– $\Gamma_f = \Gamma\,[r_0 \mapsto \Gamma(r_0)\,[n \mapsto \Sigma(r_0)]]$ (3) - hyp.3 + hyp.5

- $\Sigma_f = \Sigma$ (4) - hyp.3 + hyp.5
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Sigma(r_0) \sqcap \Sigma(r_2)$ (5) - hyp.3 + hyp.5
- $r_0 \sim_\beta r'_0$ (6) - hyp.2 + hyp.6 + (5)
- $r_2 \sim_\beta r'_2$ (7) - hyp.2 + hyp.6 + (5)
- $\Sigma'(r'_0) \sqcup \Sigma'(r'_2) \leq \sigma$ (8) - hyp.1 + hyp.6 + (6) + (7)
- $r'_2 \in \mathcal{O}rphan(\mu')$ (9) - hyp.1 + hyp.6 + (7) + *Parent Node Invariant*
- $\langle r'_0, n\rangle \in \mathcal{R}_{\#Children}(\mu')$ (10) - hyp.1 + hyp.6 + (1) + (6)
- $\mu'_f = \mu'[r'_0 \mapsto \mu'(r'_0)[n \mapsto r'_2]]$ (11) - hyp.4 + (9) + (10)
- $\Gamma'_f = \Gamma'[r'_0 \mapsto \Gamma'(r'_0)[n \mapsto \Sigma'(r'_0)]]$ (12) - hyp.4 + (9) + (10)
- $\Sigma'_f = \Sigma'$ (13) - hyp.4 + (9) + (10)
- $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta,\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$ (14) - hyp.1 + hyp.6 + (2) - (4) + (8) + (11) - (13) + + Low Object Update

Case III: $r_2 \notin \mathcal{O}rphan(\mu)$ (hyp.5) and $\Sigma(r_0) \leq \sigma$ (hyp.6). We conclude that there is a reference r_p, a memory $\hat{\mu}$, a labeling $\langle \hat{\Gamma}, \hat{\Sigma}\rangle$, such that $\langle r_p, r_2\rangle \in \mathcal{R}_{Parent}(\mu)$:

- $\langle \mu, \Gamma, \Sigma, \langle r_p, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_0) \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle\rangle \Downarrow_{rem} \langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \_, \_\rangle$ (1) - hyp.3 + hyp.5
- $\langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \Downarrow_{app} \langle \mu_f, \Gamma_f, \Sigma_f, \_, \_\rangle$ (2) - hyp.3 + hyp.5
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \leq \Sigma(r_0) \sqcap \Sigma(r_2)$ (3) - hyp.3 + hyp.5
- $r_0 \sim_\beta r'_0$ (4) - hyp.2 + hyp.6 + (3)
- $r_2 \sim_\beta r'_2$ (5) - hyp.2 + hyp.6 + (3)
- $\Sigma'(r'_2) \sqcup \Sigma'(r'_0) \leq \sigma$ (6) - hyp.2 + hyp.6 + (3)
- $\langle r'_p, r'_2\rangle \in \mathcal{R}_{Parent}(\mu')$, and $r_p \sim_\beta r'_p$, for some r'_p (7) - hyp.5 + hyp.6 + (5) + (6) + *Parent Node Invariant*
- $r'_2 \notin \mathcal{O}rphan(\mu')$ (8) - (7) + *DOM Integrity*

From hyp.4 + (8), we conclude that there is a memory $\hat{\mu}'$ and a labeling $\langle \hat{\Gamma}', \hat{\Sigma}'\rangle$:

- $\langle \mu', \Gamma', \Sigma', \langle r'_p, \_, r'_2\rangle, \langle \sigma'_0 \sqcup \Sigma'(r'_0) \sqcup \Sigma'(r'_2), \sigma'_1, \sigma'_2\rangle\rangle \Downarrow_{rem} \langle \hat{\mu}', \hat{\Gamma}', \hat{\Sigma}', \_, \_\rangle$ (9) - hyp.4 + (6)
- $\langle \hat{\mu}', \hat{\Gamma}', \hat{\Sigma}', \langle r'_0, \_, r'_2\rangle, \langle \sigma'_0, \sigma'_1, \sigma'_2\rangle\rangle \Downarrow_{app} \langle \mu'_f, \Gamma'_f, \Sigma'_f, \_, \_\rangle$ (10) - hyp.4 + (6)
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \leq \Sigma'(r'_0) \sqcap \Sigma'(r'_2)$ (11) - hyp.4 + (6)
- $\langle r_p, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_0) \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle \approx_{\beta,\sigma} \langle r'_p, \_, r'_2\rangle, \langle \sigma'_0 \sqcup \Sigma'(r'_0) \sqcup \Sigma'(r'_2), \sigma'_1, \sigma'_2\rangle$ (12) - hyp.2 + hyp.6 + (6)
- $\hat{\mu}, \hat{\Gamma}, \hat{\Sigma} \approx_{\hat{\beta},\sigma} \hat{\mu}', \hat{\Gamma}', \hat{\Sigma}'$, for some $\hat{\beta}$ that extends β (13) - hyp.1 + (1) + (9) + (12) + **NI**($\Downarrow_{rem}$) (Lemma 6)
- $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta',\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$, for some β' that extends $\hat{\beta}$ (and therefore β) (14) - hyp.2 + (2) + (10) + (13) + **NI**($\Downarrow_{app}$) (**Case II**)

Lemma 10 (Cofinement for $\Downarrow_{len}$). $\Downarrow_{len}$ *is confined.*

Proof. Since $\Downarrow_{len}$ does not modify the memory, we conclude that it is confined.

Lemma 11 (Noninterference for $\Downarrow_{len}$). NI($\Downarrow_{len}$).

Proof. Hypotheses:

- $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
- $\langle r, \_\rangle, \langle \sigma_0, \sigma_1 \rangle \approx_{\beta,\sigma} \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1 \rangle$ hyp.2
- $\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1 \rangle \rangle \ \Downarrow_{len} \ \langle \mu, \Gamma, \Sigma, n, \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \rangle$ hyp.3
- $\langle \mu', \Gamma', \Sigma', \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1 \rangle \rangle \ \Downarrow_{len} \ \langle \mu', \Gamma', \Sigma', n', \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \rangle$ hyp.4

We conclude that:

- $\langle r, n \rangle \in \mathcal{R}_{\#Children}(\mu)$ (1) - hyp.3
- $\langle r', n' \rangle \in \mathcal{R}_{\#Children}(\mu')$ (2) - hyp.4

Since the invocation of this API relation does not modify the memory, one simply has to prove that if either $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \leq \sigma$ or $\sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \leq \sigma$, then:

- $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \leq \sigma$
- $n = n'$

Without loss of generality, assume that $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \leq \sigma$ (hyp.5). We conclude that:

- $r \sim_\beta r'$ (3) - hyp.2 + hyp.5
- $\sigma'_0 \sqcup \sigma'_1 \leq \sigma$ (4) - hyp.2 + hyp.5
- $\Sigma'(r') \leq \sigma$ (5) - hyp.1 + hyp.5 + (3)
- $n = n'$ (6) - hyp.1 + hyp.5 + (1) + (2) + (3)
- $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \leq \sigma$ (7) - hyp.5 + (4) + (5)

Lemma 12 (Cofinement for $\Downarrow_{par}$). $\Downarrow_{par}$ *is confined.*

Proof. Since $\Downarrow_{par}$ does not modify the memory, we conclude that it is confined.

Lemma 13 (Noninterference for $\Downarrow_{par}$). NI($\Downarrow_{par}$).

Proof. Hypotheses:

- $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
- $\langle r, \_\rangle, \langle \sigma_0, \sigma_1 \rangle \approx_{\beta,\sigma} \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1 \rangle$ hyp.2
- $\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1 \rangle \rangle \ \Downarrow_{len} \ \langle \mu, \Gamma, \Sigma, v, \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \rangle$ hyp.3
- $\langle \mu', \Gamma', \Sigma', \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1 \rangle \rangle \ \Downarrow_{len} \ \langle \mu', \Gamma', \Sigma', v', \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \rangle$ hyp.4
- $\langle r_p, r \rangle \in \mathcal{R}_{Parent}(\mu) \Rightarrow v = r_p$ and $r \in \mathcal{O}rphan(\mu) \Rightarrow v =$ *undefined* hyp.5
- $\langle r'_p, r' \rangle \in \mathcal{R}_{Parent}(\mu') \Rightarrow v' = r'_p$ and $r' \in \mathcal{O}rphan(\mu') \Rightarrow v' =$ *undefined* hyp.6

Without loss of generality, assume that $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \leq \sigma$ (hyp.7). We conclude that:

- $r \sim_\beta r'$ (1) - hyp.2 + hyp.7
- $\sigma'_0 \sqcup \sigma'_1 \leq \sigma$ (2) - hyp.2 + hyp.7
- $\Sigma'(r') \leq \sigma$ (5) - hyp.1 + hyp.7 + (1)

– $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \sigma_0' \sqcup \sigma_1' \sqcup \Sigma'(r') \leq \sigma$ (6) - hyp.7 + (2) + (5)

We now proceed by case analysis.

Case I: $\langle r_p, r\rangle \in \mathcal{R}_{Parent}(\mu)$ (hyp.8). We conclude that:

– $\langle r_p', r'\rangle \in \mathcal{R}_{Parent}(\mu')$ and $r_p \sim_\beta r_p'$ (7) - hyp.1 + hyp.7 + hyp.8 + (1) + *Parent Node Invariant*
– $v \sim_\beta v'$ (8) - hyp.5 + hyp.6 + hyp.8 + (7)

Case II: $r \in \mathcal{O}rphan(\mu)$ (hyp.8). We conclude that:

– $r \in \mathcal{O}rphan(\mu)$ (7) - hyp.1 + hyp.7 + hyp.8 + (1) + *Parent Node Invariant*
– $v \sim_\beta v'$ (8) - hyp.5 + hyp.6 + hyp.8 + (7)

Lemma 14 (Cofinement for $\Downarrow_{ind}$). *$\Downarrow_{ind}$ is confined.*

Proof. Since $\Downarrow_{ind}$ does not modify the memory, we conclude that it is confined.

Lemma 15 (Noninterference for $\Downarrow_{ind}$). **NI**($\Downarrow_{ind}$).

Proof. Hypotheses:

– $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
– $\langle r, i\rangle, \langle \sigma_0, \sigma_1\rangle \approx_{\beta,\sigma} \langle r', i'\rangle, \langle \sigma_0', \sigma_1'\rangle$ hyp.2
– $\langle \mu, \Gamma, \Sigma, \langle r, i\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \Downarrow_{ind} \langle \mu, \Gamma, \Sigma, v, \sigma_0 \sqcup \sigma_1 \sqcup \hat{\sigma}\rangle$ hyp.3
– $\langle \mu', \Gamma', \Sigma', \langle r', i'\rangle, \langle \sigma_0', \sigma_1'\rangle\rangle \Downarrow_{ind} \langle \mu', \Gamma', \Sigma', v', \sigma_0' \sqcup \sigma_1' \sqcup \hat{\sigma}'\rangle$ hyp.4
– $i \in dom(\mu(r)) \Rightarrow v = \mu(r)(i) \wedge \hat{\sigma} = \Gamma(r)(i)$ hyp.5
– $i \notin dom(\mu(r)) \Rightarrow v = \mathit{undefined} \wedge \hat{\sigma} = \Sigma(r)$ hyp.6
– $i' \in dom(\mu'(r')) \Rightarrow v' = \mu'(r')(i') \wedge \hat{\sigma}' = \Gamma'(r')(i')$ hyp.7
– $i' \notin dom(\mu'(r')) \Rightarrow v' = \mathit{undefined} \wedge \hat{\sigma}' = \Sigma'(r')$ hyp.8

Assume, without loss of generality, that $\sigma_0 \sqcup \sigma_1 \leq \sigma$ (hyp.9). We conclude that:

– $r \sim_\beta r'$ (1) - hyp.2 + hyp.9
– $i = i'$ (2) - hyp.2 + hyp.9
– $\sigma_0' \sqcup \sigma_1' \leq \sigma$ (3) - hyp.2 + hyp.9

We proceed by case analysis.

Case I: $i \in dom(\mu(r))$ (hyp.10). Assume, without loss of generality, that $\Gamma(r)(i) \leq \sigma$ (hyp.11). We conclude that:

– $i \in dom(\mu'(r'))$, $\Gamma'(r')(i) \leq \sigma$, $\mu(r)(i) \sim_\beta \mu'(r')(i)$ (4) - hyp.1 + hyp.10 + hyp.11 + (1)
– $v \sim_\beta v'$ (5) - hyp.5 + hyp.7 + hyp.10 + (4)
– $\sigma_0 \sqcup \sigma_1 \sqcup \Gamma(r)(i) \sqcup \sigma_0' \sqcup \sigma_1' \sqcup \Gamma'(r')(i) \leq \sigma$ (6) - hyp.9 + hyp.11 + (3) + (4)

Case II: $i \notin dom(\mu(r))$ (hyp.10). Assume, without loss of generality, that $\Sigma(r) \leq \sigma$ (hyp.11). We conclude that:

- $dom(\mu(r)) = dom(\mu'(r'))$ and $\Sigma'(r') \leq \sigma$ (4) - hyp.1 + hyp.11 + (1)
- $i \notin dom(\mu'(r'))$ (5) - hyp.10 + (4)
- $v = \mathit{undefined}$ (6) - hyp.6 + hyp.10
- $v' = \mathit{undefined}$ (7) - hyp.8 + (5)
- $v \sim_\beta v'$ (8) - (6) + (7)
- $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \leq \sigma$ (6) - hyp.9 + hyp.11 + (3) + (4)

Lemma 16 (Cofinement for $\Downarrow_{sib}$). *$\Downarrow_{ind}$ is confined.*

Proof. Since $\Downarrow_{sib}$ does not modify the memory, we conclude that it is confined.

Lemma 17 (Noninterference for $\Downarrow_{sib}$). NI($\Downarrow_{ind}$).

Proof. Hypotheses:

- $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$ hyp.1
- $\langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle \approx_{\beta,\sigma} \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1\rangle$ hyp.2
- $\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \Downarrow_{sib} \langle \mu, \Gamma, \Sigma, v, \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \hat{\sigma}\rangle$ hyp.3
- $\langle \mu', \Gamma', \Sigma', \langle r', \_\rangle, \langle \sigma'_0, \sigma'_1\rangle\rangle \Downarrow_{sib} \langle \mu', \Gamma', \Sigma', v', \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \sqcup \hat{\sigma}'\rangle$ hyp.4

Assume, without loss of generality, that: $\sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r) \sqcup \hat{\sigma} \leq \sigma$ (hyp.5), where $\hat{\sigma}$ depends on the rule used in the derivation of hyp.1. We conclude that:

- $r \sim_\beta r'$ (1) - hyp.2 + hyp.5
- $\sigma'_0 \sqcup \sigma'_1 \leq \sigma$ (2) - hyp.2 + hyp.5
- $\Sigma'(r') \leq \sigma$ (3) - hyp.1 + hyp.5 + (1)

We proceed by case analysis.

Case I: $\langle r_p, r\rangle \in \mathcal{R}_{Parent}(\mu)$ (hyp.6), for some reference r_p. We conlcude that there are two integers i and n such that:

- $\langle r_p, n\rangle \in \mathcal{R}_{\#Children}(\mu)$ (4) - hyp.3 + hyp.6
- $\mu(r_p)(i) = r$ (5) - hyp.3 + hyp.6
- $\langle r'_p, r'\rangle \in \mathcal{R}_{Parent}(\mu')$ and $r_p \sim_\beta r'_p$ (6) - hyp.1 + hyp.5 + hyp.6 + (1) + *Parent Node Invariant*
- **Case I.1:** $i + 1 < n$ (hyp.7):
  - $\hat{\sigma} = \Gamma(r_p)(i+1) \leq \sigma$ (7) - hyp.3 + hyp.5 + hyp.7
  - $\Gamma(r_p)(i) \leq \sigma$ (8) - (7) + *Indexes Invariant*
  - $\Gamma'(r'_p)(i+1) \leq \sigma$ (9) - hyp.1 + (6) + (7)
  - $\Gamma'(r'_p)(i) \leq \sigma$ (10) - (9) + *Indexes Invariant*
  - $\mu(r_p)(i) \sim_\beta \mu'(r'_p)(i)$ (11) - hyp.1 + (6) + (8)
  - $v = \mu(r_p)(i+1) \sim_\beta \mu'(r'_p)(i+1) = v'$ (12) - hyp.1 + hyp.3 + hyp.4 + (6) + (7)
  - $\hat{\sigma}' = \Gamma'(r'_p)(i+1) \leq \sigma$ (13) - hyp.1 + (6) + (8)
  - $\sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \sqcup \hat{\sigma}' \leq \sigma$ (14) - (2) + (3) + (13)
- **Case I.2:** $i + 1 \geq n$ (hyp.7):
  - $\hat{\sigma} = \Sigma(r_p) \leq \sigma$ (7) - hyp.3 + hyp.5 + hyp.7

- $\Gamma(r_p)(i) \leq \sigma$ (8) - (7) + *Indexes Invariant*
- $\Gamma'(r'_p)(i) \leq \sigma$ (9) - hyp.1 + (6) + (8)
- $r = \mu(r_p)(i) \sim_\beta \mu'(r'_p)(i) = r'$ (10) - hyp.1 + (6) + (8)
- $\Sigma'(r'_p) \leq \sigma$ (11) - hyp.1 + (6) + (7)
- $\langle r'_p, n\rangle \in \mathcal{R}_{\#Children}(\mu')$ (12) - hyp.1 + (4) + (6) + (7)
- $\hat{\sigma}' = \Sigma'(r'_p) \leq \sigma$ (13) - hyp.4 + (6) + (11) + (12)
- $v = \mathit{undefined}$ (14) - hyp.3 + hyp.6 + hyp.7
- $v' = \mathit{undefined}$ (15) - hyp.4 + (6) + (12)
- $v \sim_\beta v'$ (16) - (14) + (15)
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \sqcup \hat{\sigma}' \leq \sigma$ (17) - (2) + (3) + (13)

Case II: $r \in \mathcal{O}rphan(\mu)$ (hyp.6). We conlcude that:

- $v = \mathit{undefined}$ (4) - hyp.3 + hyp.6
- $r' \in \mathcal{O}rphan(\mu')$ (5) - hyp.5 + (1) + *Parent Node Invariant*
- $v' = \mathit{undefined}$ (6) - hyp.4 + (5)
- $\hat{\sigma} = \hat{\sigma}' = \bot$ (7) - hyp.3 + hyp.4 + hyp.6 + (5)
- $\sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'(r') \sqcup \hat{\sigma}' \leq \sigma$ (17) - (2) + (3) + (7)

Lemma 18 (Confiment for the DOM API). *$\mathcal{R}^{DOM}_{API}$ is confined.*

Proof. Corollary of Lemmas 2, 5, 8, 10, 12, 14, and 16.

Lemma 19 (Preservation of the *Indexes Invariant*). *Let $\Downarrow_{API}$ be an API relation in $rng(\mathcal{R}^{DOM}_{API})$, μ a memory well-labeled by $\langle \Gamma, \Sigma\rangle$, $\overrightarrow{v}$ a sequence of values respectively labeled by the security levels in the sequence $\overrightarrow{\sigma}$, such that: (1) $\langle \mu, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma}\rangle \Downarrow_{API} \langle \mu', \Gamma', \Sigma', v', \sigma'\rangle$ and (2) $\langle \mu, \Gamma, \Sigma\rangle \in \mathcal{WL}^{DOM}_{indexes}$. Then, $\langle \mu', \Gamma', \Sigma'\rangle \in \mathcal{WL}^{DOM}_{indexes}$.*

Proof. We proceed by case analysis. The only DOM APIs that change the indexes of DOM Element objects are: $\Downarrow_{rem}$ and $\Downarrow_{app}$. Hence, it follows that:

- $\overrightarrow{v} = \langle r_0, \_, r_2\rangle$, for two references r_0 and r_2 (1) - hyp.1
- $\overrightarrow{\sigma} = \langle \sigma_0, \_, \sigma_2\rangle$, for two security levels σ_0 and σ_2 (2) - hyp.1

Case I: $\Downarrow_{API}$=$\Downarrow_{rem}$ (hyp.3). We conclude that there are two integers i and n, an object o_0, and a labeling object γ_0, such that:

- $\mu(r_0)(i) = r_2$ and $\langle r_0, n+1\rangle \in \mathcal{R}_{\#Children}(\mu)$ (3) - hyp.1 + hyp.3 + (1) + (2)
- $dom(o_0) = dom(\gamma_0) = dom(\mu(r_0))\backslash\{n\}$ (4) - hyp.1 + hyp.3 + (1) + (2)
- $\forall_{0\leq j<i} \cdot o_0(j) = \mu(r_0)(j)$, $\forall_{i\leq j<n} \cdot o_0(j) = \mu(r_0)(j+1)$, $o_0(@tag) = \mu(r_0)(@tag)$ (5) - hyp.1 + hyp.3 + (1) + (2)
- $\forall_{0\leq j<i} \cdot \gamma_0(j) = \Gamma(r_0)(j)$, $\forall_{i\leq j<n} \cdot \gamma_0(j) = \Gamma(r_0)(j+1)$, $\gamma_0(@tag) = \Gamma(r_0)(@tag)$ (6) - hyp.1 + hyp.3 + (1) + (2)
- $\mu' = \mu\,[r_0 \mapsto o_0]$, $\Gamma' = \Gamma\,[r_0 \mapsto \gamma_0]$, and $\Sigma' = \Sigma$ (7) - hyp.1 + hyp.3 + (1) + (2)
- $\forall_{0\leq j<n}\Gamma(r_0)(j) \leq \Gamma(r_0)(j+1)$ and $\Gamma(r_0)(n) \leq \Sigma(r_0)$ (8) - hyp.2 + (3)

– $\forall_{0\leq j<n-1} \Gamma'(r_0)(j) \leq \Gamma'(r_0)(j+1)$ and $\Gamma'(r_0)(n-1) \leq \Sigma(r_0)$ (9) - (6) - (8)
– $\langle \mu', \Gamma', \Sigma' \rangle \in \mathcal{WL}^{DOM}_{indexes}$ (10) - hyp.2 + (7) + (9)

Case II: $\Downarrow_{API}$=$\Downarrow_{app}$ (hyp.3) and $r_2 \in \mathcal{O}rphan(\mu)$ (hyp.4). We conclude that there are is an integer n, such that:

– $\langle r_0, n \rangle \in \mathcal{R}_{\#Children}(\mu)$ (3) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\mu' = \mu\,[r \mapsto \mu(r_0)\,[n \mapsto r_2]]$ (4) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\Gamma' = \Gamma\,[r_0 \mapsto \Gamma(r_0)\,[n \mapsto \Sigma(r_0) \sqcup \Sigma(r_2)]]$ (5) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\Sigma' = \Sigma$ (6) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\forall_{0\leq j<n-1} \Gamma(r_0)(j) \leq \Gamma(r_0)(j+1)$ and $\Gamma(r_0)(n-1) \leq \Sigma(r_0)$ (7) - hyp.2 + (3)
– $\forall_{0\leq j<n} \Gamma'(r_0)(j) \leq \Gamma'(r_0)(j+1)$ and $\Gamma'(r_0)(n) \leq \Sigma'(r_0)$ (8) - (5) - (7)
– $\langle \mu', \Gamma', \Sigma' \rangle \in \mathcal{WL}^{DOM}_{indexes}$ (9) - hyp.2 + (4) - (6) + (8)

Case III: $\Downarrow_{API}$=$\Downarrow_{app}$ (hyp.3) and $r_2 \notin \mathcal{O}rphan(\mu)$ (hyp.4). We conclude that there is a reference r_p, a memory $\hat{\mu}$, a labeling $\langle \hat{\Gamma}, \hat{\Sigma} \rangle$, such that $\langle r_p, r_2 \rangle \in \mathcal{R}_{Parent}(\mu)$:

– $\langle \mu, \Gamma, \Sigma, \langle r_p, \_, r_2 \rangle, \langle \sigma_0 \sqcup \Sigma(r_0) \sqcup \Sigma(r_2), \sigma_1, \sigma_2 \rangle \rangle \;\Downarrow_{rem}\; \langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \_, \_ \rangle$ (3) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma}, \langle r_0, \_, r_2 \rangle, \langle \sigma_0, \sigma_1, \sigma_2 \rangle \rangle \;\Downarrow_{app}\; \langle \mu', \Gamma', \Sigma', \_, \_ \rangle$ (4) - hyp.1 + hyp.3 + hyp.4 + (1) + (2)
– $\langle \hat{\mu}, \hat{\Gamma}, \hat{\Sigma} \rangle \in \mathcal{WL}^{DOM}_{indexes}$ (5) - hyp.2 + **Case I**
– $\langle \mu', \Gamma', \Sigma' \rangle \in \mathcal{WL}^{DOM}_{indexes}$ (6) - (5) + **Case II**