An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript

José Fragoso Santos and Tamara Rezk

Inria
{firstname.lastname}@inria.fr

Abstract. Web application designers and users alike are interested in isolation properties for trusted JavaScript code in order to prevent confidential resources from being leaked to untrusted parties. Noninterference provides the mathematical foundation for reasoning precisely about the information flows that take place during the execution of a program. Due to the dynamicity of the language, research on mechanisms for enforcing noninterference in JavaScript has mostly focused on dynamic approaches. We present the first information flow monitor inlining compiler for a realistic core of JavaScript. We prove that the proposed compiler enforces termination-insensitive noninterference and we provide an implementation that illustrates its applicability.

1 Introduction

Client-side JavaScript programs often include untrusted code dynamically loaded from third-party code providers, such as online advertisers. This issue raises the need for enforcement mechanisms that isolate trusted code from code that comes from untrusted sources. Such mechanisms must prevent trusted programs from leaking confidential resources. Noninterference [13] is an expressive and elegant property that formally defines secure information flow, thus being commonly used as a soundness criteria for dynamic and static analyses that aim at enforcing secure information flow.

Due to the dynamic nature of JavaScript, research on mechanisms to check the compliance of JavaScript programs with noninterference has mostly focused on dynamic approaches. In practice, there are two main approaches for implementing a JavaScript information flow monitor: either one modifies a JavaScript engine so that it additionally implements the security monitor (as in [9]), or one inlines the monitor in the original program (as in [7, 11]). The second approach, which we follow, has the advantage of being *browser-independent*. We present the first compiler that inlines an information flow monitor for a subset of JavaScript that we call Core JavaScript. Core JavaScript includes the main standard features of the language, such as objects with prototypical inheritance and closures, as well as non-standard features, such as several unusual ways for interacting with the *global object* – a special object that binds global variables.

The proposed compiler is proven sound w.r.t. a standard definition of input-output termination insensitive noninterference for monitors. In this setting, attackers are assumed to be unable to observe the contents of intermediate memory

N. Cuppens-Boulahia et al. (Eds.): SEC 2014, IFIP AICT 428, pp. 278–292, 2014.

states or use divergent executions as a means of disclosing confidential resources. Informally, we prove that the execution of a compiled program only goes through if it is noninterferent; otherwise, the constraints inlined in the program by the compiler cause it to diverge. The paper is divided into two main sections. Section 2 presents an information flow monitored semantics for Core JavaScript that is proven *sound*, i.e. proven to enforce termination-insensitive noninterference. Section 3 presents an inlining compiler that rewrites Core JavaScript programs in order to simulate their execution in the monitor. The compiler is proven *correct*, meaning that the execution of a program goes through in the monitor *if and only if* the execution of its instrumentation by the inlining compiler goes through in the original semantics. We have implemented a prototype of the proposed compiler, which is available via at [1] together with a broad set of examples and a full version of this paper that includes the proofs of the main theorems.

1.1 Core JavaScript Syntax and Semantics

The syntax of Core JavaScript is given in Figure 1. Some expressions are annotated with one or two unique indexes for use by the compiler, which are omitted when not needed. In the examples, we use $o.p$ as an abbreviation for o["p"]. Objects are the fundamental data type in JavaScript. Informally, an object can be seen as a collection of named values. At the semantic level, we model objects as partial functions from strings, taken from a set $\mathcal{S}tr$, to values. JavaScript values comprise: (1) primitive values (taken from a set $\mathcal{P}rim$), (2) object references (taken from a set $\mathcal{R}ef$), and (3) parsed function literals (for which we use the lambda notation: $\lambda x.\mathsf{var}\ y_1, \cdots, y_n; e$). $\mathcal{P}rim$ includes strings, numbers, and booleans, as well as two special values *null* and *undefined*. The strings in the domain of an object are called its *properties*. Some properties are internal and therefore can neither be changed nor read by programs. For clarity, these properties are prefixed with an "@". Every expression that creates an object in memory yields a free non-deterministically chosen reference that points to it. Hence, references can be viewed as pointers to objects. Given an object o, we use $\#o$ to denote the reference that points to it. Finally, a *memory* μ is a partial mapping from references to objects.

Notation. 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.

Function Calls and Variables. As in JavaScript, we model scope *via scope objects* [2, 10]. Every function call triggers the creation of a scope object which maps its formal parameter as well as the variables declared in its body to their corresponding values. A scope object is said to be *active* if it is associated with the function that is currently executing. Furthermore, every scope object defines

$e ::=$	v^i	value	$\mid \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}$	function literal
$\mid$	this^i	this keyword	$\mid \{\}^i$	object literal
$\mid$	$e_0\ \mathsf{op}^i\ e_1$	binary operation	$\mid e_0(e_1)^i$	function call
$\mid$	x^i	variable	$\mid e_0[e_1](e_2)^i$	method call
$\mid$	$x = e$	variable assignment	$\mid e_0,\ e_1$	sequence
$\mid$	$e_0[e_1]^i$	property look-up	$\mid e_0\ ?^{i,j}\ (e_1) : (e_2)$	conditional
$\mid$	$e_0[e_1] = e_2$	property assignment		

e, e_0, e_1 and e_2 represent expressions, i and j represent program indexes, x, y_1, ..., y_n represent variable names, and op represents binary operators.

Fig. 1. Syntax of Core JavaScript

a property @*scope* that points to the scope object that was active when the corresponding function literal was evaluated.

The sequence of scope objects that can be accessed from a given scope object through the respective @*scope* properties is called a *scope-chain*. The *global object*, which is assumed to be pointed by a fixed reference *#glob*, is the object that is at the end of every scope-chain and therefore it is the object that binds *global variables*. In order to determine the value associated with a given variable, one has to inspect all objects in the scope-chain that starts in the *active* scope object. This behavior is modeled by the semantic relation $\mathcal{R}_{Scope}\text{İf}\ \langle \mu, r_0, x\rangle\ \mathcal{R}_{Scope}\ r_1$, then r_1 is the reference that points to the scope object that is closest to the one pointed by r_0 in the corresponding scope-chain (whose objects are in the range of μ) and which defines a binding for variable x.

Function Literals and Variable Assignments. The evaluation of a function literal yields a reference to an object, called *a function object*, that stores its parsed counterpart. More specifically, since every function is executed in the environment in which the corresponding function literal was evaluated, every function object defines the following two properties: (1) @*code* that stores the parsed function literal and (2) *@fscope* that stores the reference that points to the scope object that was active when the corresponding function literal was evaluated. Assuming that the global object defines a variable *out* originally set to *null*, the evaluation of the program presented below on the left yields the value 0 and creates in memory the list of objects displayed below on the right:

$$
\begin{array}{ll}
(\mathsf{function}(x)\{ & o_s^0 = [@scope \mapsto \#glob, x \mapsto 0, g \mapsto o_g, h \mapsto o_h] \\
\quad \mathsf{var}\ g, h; & o_s^g = \left[@scope \mapsto \#o_s^0, x \mapsto 1\right] \\
\quad g = \mathsf{function}(x)\{h(2)\}, & o_s^h = \left[@scope \mapsto \#o_s^0, y \mapsto 2\right] \\
\quad h = \mathsf{function}(y)\{out = x\}, & o_0 = [@code \mapsto \lambda x.\mathsf{var}\ g, h; \hat{e}, @fscope \mapsto \#glob] \\
\quad g(1) & o_g = \left[@code \mapsto \lambda x.h(2), @fscope \mapsto \#o_s^0\right] \\
\})(0); & o_h = \left[@code \mapsto \lambda y.out = x, @fscope \mapsto \#o_s^0\right]
\end{array}
$$

where (1) o_s^0, o_s^g, and o_s^h correspond to the scope objects associated with the invocation of the anonymous function, of function g, and of function h, respectively, (2) objects o_0, o_g, and o_h correspond to their respective function objects, and (3) $\hat{e}$ corresponds to the body of the anonymous function. After the execution of this program, the global object maps *out* to 0 and not to 1, because the

scope object that is closest to o_s^h and which defines a binding for x is o_s^0 and not o_s^g (which does not belong to the scope-chain of o_s^h).

Object Literals and Property Look-ups. Core JavaScript features prototypical inheritance. This means that every object (except scope objects and function objects) defines a property *_prot_* that stores a reference to its prototype. When trying to look-up the value of a property p of an object o, the semantics first checks whether $p \in dom(o)$. If $p \in dom(o)$, the property look-up yields $o(p)$, otherwise the semantics checks whether the prototype of o defines a property named p, and so forth. The sequence of objects that can be accessed from a given object through the respective *_prot_* properties is called a *prototype-chain*. The prototype-chain inspection procedure is emulated by the semantic relation $\mathcal{R}_{Proto}$. If $\langle \mu, r, m, \Gamma, \Sigma \rangle \; \mathcal{R}_{Proto} \; \langle r', \sigma \rangle$, then r' is the closest reference to r in its corresponding prototype-chain (whose objects are in the range of μ) that defines a binding for m (we ignore, by now, the remaining elements of the relation, since they are used by the monitored semantics and not by the original semantics). The evaluation of an object literal yields a free non-deterministically chosen reference that points to a new object that only defines a property *_prot_* originally set to *null*. Hence, the evaluation of $o_0 = \{\},\ o_0.p = 0,\ o_1 = \{\},\ o_1.\_prot\_ = o_0,\ o_1.p$ yields 0, because, although o_1 does not define property p, its prototype does. When looking-up the value of a property p in an object o, if p is not defined in the whole prototype-chain of o, instead of yielding an error, the semantics yields *undefined*. Therefore, the expression $o = \{\}, o.p$ evaluates to *undefined*.

Method Calls and the this *Keyword.* Functions whose references are assigned to properties of an object are called its *methods*. A function can be either invoked as a normal function or as a method. When calling a function as a method, the this keyword is bound to the corresponding object, otherwise it is bound to the global object. Therefore, every scope object defines a property $@this$ (that was omitted in the first example) that holds the value of the this keyword in that scope. We further remark that given an object o, every method m accessible from o through its prototype-chain can be called as a method of o. Hence, suppose that in a memory μ, the global object defines two variables o_0 and o_1 that hold references to $[\_prot\_ \mapsto null, f \mapsto \#o_f]$ and $[\_prot\_ \mapsto \#o_0]$ respectively, where $\#o_f$ is the reference of a given function object. In the evaluation of expression $o_1.f(0)$, the semantics starts by creating a scope object in which property $@this$ is set to $\#o_1$ and then proceeds with the evaluation of the body of f.

The remaining program constructs have the usual semantics, which can be understood from the formal definition. We make use of a big-step semantics for Core JavaScript with the following shape: $r \vdash \langle \langle \mu, e \rangle \rangle \Downarrow \langle \langle \mu', v \rangle \rangle$, where r is the reference of the active scope object, μ and μ' are the initial and final memories respectively, e the expression to be evaluated, and v the value to which it evaluates. Due to space constraints we choose not to give its formal definition here. Instead, we only present its monitored version, $\Downarrow_{IF}$ (Figure 2). In order to obtain $\Downarrow$ from $\Downarrow_{IF}$, one simply has to remove from $\Downarrow_{IF}$ the monitor constraints.

2 Monitoring Secure Information Flow

Specifying Security Policies. The specification of security policies usually relies on two key elements: a lattice of security levels and a labeling that maps resources to security levels. In the examples, we use $\mathcal{L} = \{H, L\}$ with $L \leq H$, meaning that resources labeled with level L (low) are less confidential than resources labeled with H (high). Hence, after the execution of a program, resources labeled with H are allowed to depend on resources originally labeled with L, but not the opposite, since that would entail an information leak. In the following, we always assume that $\leq$ and $\sqcup$ correspond to the order relation and the least upper bound on security levels respectively. A security labeling is a pair $\langle \Gamma, \Sigma \rangle$ where $\Gamma : \mathcal{R}ef \to \mathcal{S}tr \to \mathcal{L}$ maps each property in every object to a security level and $\Sigma : \mathcal{R}ef \to \mathcal{L}$ maps every reference to the *structure security level* [9] of the corresponding object. Hence, given an object o pointed to by a reference r_o, $\Gamma(r_o)(p)$ is the security level associated with o's property p and $\Sigma(r_o)$ is the structure security level of o. Notice that, as every variable is modeled as a property of a given scope object, Γ also maps variables to their corresponding security levels, treating variables and properties uniformly. In the examples, we assume that variables h and l are respectively labeled with levels H and L. The structure security level of an object can be understood as the security level associated with its domain. The need to associate a security level with the domain of every object arises because it is possible for a program to leak information *via* the domain of an object. For instance, after the evaluation of $o = \{\},\ h\ ?\ (o.p = 0) : (null),\ l = o.p$, the final value of the low variable l depends on the initial value of the high variable h. Precisely, when $h \in \{false, 0, null, undefined\}$, property p is not added to the domain of o and l is set to *undefined*, whereas in all other cases, both property p and variable l are set to 0. Finally, we observe that initial memories are assumed to include a global object for the binding of global variables. Accordingly, initial labellings apply both to the global object as well as the objects that are initially accessible through the global object.

Low-Equality. We introduce a notion of indistinguishability between memories that models the "power" of an attacker that can only observe resources up to a given security level σ, called low-equality, denoted by $\approx_{\beta,\sigma}$. Informally, two labeled memories are low-equal at level σ if they coincide in the resources labeled with levels $\leq \sigma$. Since references are non-deterministically chosen we need to be able to relate observable references in two different memories. To this end, we parametrize the low-equality relation with a partial injective function $\beta : \mathcal{R}ef \to \mathcal{R}ef$ [5] that relates observable references. The low-equality definition relies on a binary relation on values, named β-equality and denoted by $\sim_\beta$. *β-Equality:* two objects are β-equal if they have the same domain and all their properties are β-equal, primitive values and parsed functions are β-equal if they are equal, and two references r_0 and r_1 are β-equal if $\beta(r_0) = r_1$.

In the following, given a property labeling Γ, a reference r, and security level σ, we denote by $\Gamma(r)|_\sigma$, the set of observable properties in $\Gamma(r)$ at level σ: $\Gamma(r)|_\sigma = \{p \mid \Gamma(r)(p) \leq \sigma\}$.

Definition 1 (Low-Equality $\approx_{\beta,\sigma}$). *Two memories μ_0 and μ_1 are said to be low equal with respect to $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, security level σ, and function β, written $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$, iff 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: $\Gamma_0(r_0)|_\sigma = \Gamma_1(r_1)|_\sigma = P$. **(2)** The objects coincide in their observable domains: $\mu_0(r_0)|_P \sim_\beta \mu_1(r_1)|_P$. **(3)** Either the domains of both objects are not observable, or they are both observable and completely coincide: $(\Sigma_0(r_0) \leq \sigma \vee \Sigma_1(r_1) \leq \sigma) \Rightarrow \Sigma_0(r_0) \sqcup \Sigma_1(r_1) \leq \sigma \wedge dom(\mu_0(r_0)) = dom(\mu_1(r_1))$.*

Monitored Semantics. The rules of the monitored semantic relation, $\Downarrow_{IF}$, defined in Figure 2, have the form $r, \sigma_{pc} \vdash \langle \mu, e, \Gamma, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Gamma', \Sigma', \sigma \rangle$, where σ_{pc} is the security level of the execution context, $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$ are the initial and final labellings, and σ is the *reading effect* of e [13]. The remaining elements keep their original meaning in $\Downarrow$. The *reading effect* of an expression is defined as the least upper bound on (1) the levels of the resources on which the value to which it evaluates depends and (2) the level of the current context, σ_{pc}. The monitored execution of an expression e can be interpreted as an extension of the unmonitored execution of e that additionally performs the *abstract execution* of e on the abstract memory given by Γ. Hence, the computation of Γ' and σ precisely mirrors the computation of μ' and v. The monitored semantics makes use of a relation $\mathcal{R}_{NewScope}$, which models the storing of a new scope object in memory. Hence, if $\langle \mu, \Gamma, \Sigma, r_f, v_{arg}, r_{this}, \sigma_{pc}, \sigma_{arg} \rangle \; \mathcal{R}_{NewScope} \; \langle \mu', e, \Gamma', \Sigma', r', \sigma'_{pc} \rangle$, then: (1) μ' and $\langle \Gamma', \Sigma' \rangle$ are the memory and labeling obtained from μ and $\langle \Gamma, \Sigma \rangle$ by the allocation of the new scope object in the free reference r'; (2) r_f is the reference to the function that is going to be executed, e its body, v_{arg} the argument to be used, σ_{pc} the level of the context in which the function was invoked, σ_{arg} the reading effect of the actual argument, and r_{this} the reference to the object to be used as this; and (3) σ'_{pc} is the security level at which the execution of e takes place.

Among the possible techniques to design a purely dynamic sound information flow monitor, we choose to follow the *no-sensitive-upgrade* discipline [3]. Essentially, the monitor blocks executions that try to upgrade the value of low resources within high contexts. To illustrate the idea of this strategy, consider the following program: $h \; ? \; (l = 0) : (null)$. Suppose that the monitor allows the execution of this program to go through in an initial memory that maps h to 1 just raising the level of l to H (which constitutes a sensitive upgrade). If this same program is executed in a memory that maps h to 0; in the final memory, l is labeled with L and therefore it is visible. Hence, after executing this program starting from two indistinguishable memories, we obtain two memories that are distinguishable by an attacker at level L, meaning that the attacker has learned something about the confidential resources of the program.

Function/Method Calls, Conditional Expressions, and Function Literals. The only non-trivial part concerning the monitoring of these four types of expressions has to do with how the first three update the level of the execution context in which their subexpressions are evaluated. Observe that σ_{pc} must always be higher

than or equal to the security levels of the resources that were used to decide: (1) which branch to take in a conditional expression whose code is still executing and (2) which function/method to execute in a function/method call expression whose execution is still being performed. E.g., consider the following expression:

$$f_1 = \mathsf{function}(x)\{l = 0\}, f_2 = \mathsf{function}(x)\{l = 1\}, h\ ?\ (f = f_1) : (f = f_2), f() \quad (1)$$

Since the final value of the low variable l depends on the original value of the high variable h, this program does not abide by the security policy and is therefore considered *illegal*. Hence, independently of the branch taken in the execution of the conditional, in the evaluation of the corresponding expression, the monitor must be aware that the decision to take that branch depends on the value of a high variable. Analogously, when executing the body of the function assigned to f, the monitor must be aware that the fact that it is executing that function and not another does also depend on the value of a high variable. Hence, σ_{pc} must be upgraded to high both during the execution of the taken branch of the conditional and during the execution of the body of the function bound to f. Additionally, σ_{pc} must also take into account the level of the context in which the function literal corresponding to the function that is currently evaluating was itself evaluated. Consider the expression:

$$f = h\ ?\ (\mathsf{function}(x)\{l = 0\}) : (\mathsf{function}(x)\{l = 1\}), f() \quad (2)$$

This program is illegal because after its execution, depending on the value of the high variable h, the low variable l can be either 0 or 1. To account for this type of leak, when a function literal is evaluated the level of the current context is stored in $\Gamma(r_f)(@fscope)$. Hence, every time the corresponding function is called, it is executed in a context whose level is set to be $\geq \Gamma(r_f)(@fscope)$.

Variable Assignments and Property Updates. In accordance with the no-sensitive-upgrade discipline, the monitor only allows a variable x (or a property p of an object o) to be upgraded in a context whose level is lower than or equal to its current level: $\sigma_{pc} \leq \Gamma(r_{pc})(x)$ (or $\sigma_{pc} \leq \Gamma(\#o)(p)$). Therefore, considering the expression given in Code Snippet (1), if f is a high variable, the assignments inside the branches of the conditional are allowed to go through. However, the assignment inside the body of the function bound to f is not, because the value of the execution context is high, whereas the level of the variable that is being updated is low. Notice, however, that, in the Rule [Property Assignment] (for the case in which the property to be assigned is defined), the constraint is not $\sigma_{pc} \leq \Gamma(\#o)(p)$, but instead $\sigma_0 \sqcup \sigma_1 \leq \Gamma(\#o)(p)$. Observe that, since the monitor ensures that $\sigma_{pc} \leq \sigma_0$ and $\sigma_{pc} \leq \sigma_1$, the latter constraint subsumes the former. The need for this stricter constraint arises from the fact that in a property assignment, the assignment that actually takes place depends on the reading effects of (1) the expression that evaluates to the reference of the object of the property to be assigned and (2) the expression that evaluates to the actual property whose value is to be updated. Suppose, for instance, that variable o holds an object only containing low properties. Then, even if σ_{pc} is low, the

expression $o[h] = 0$ is illegal, because depending on the value of h, it updates the value of a different low property. One cannot simply upgrade the level of the property to which h evaluates to H because that would constitute a sensitive upgrade, since for different values of h, an attacker at level L would see different properties disappearing from the observable domain of o.

Property Look-ups, Property Creation Expressions, and Object Literals. When a program looks up the value of a property p in an object o, if $p \notin dom(o)$, the security level associated with the property look-up expression must be equal to or higher than the structure security level of o, because this property look-up leaks information about its domain. In fact, since every property look-up searches the prototype-chain of the corresponding object, the security monitor has to take into account the structure security level as well as the level of property $\_prot\_$ of every object traversed during the prototype-chain inspection procedure (which corresponds to σ in $\langle \mu, r, m, \Gamma, \Sigma \rangle \; \mathcal{R}_{Proto} \; \langle r', \sigma \rangle$). For example, given a memory:

$$\mu = [\#o_0 \mapsto [p \mapsto 1, \_prot\_ \mapsto null], \#o_1 \mapsto [\_prot\_ \mapsto \#o_0], \#glob \mapsto [o_1 \mapsto \#o_1]] \quad (3)$$

and a labeling $\langle \Gamma, \Sigma \rangle$, such that Γ maps all properties in every object in the range of μ to L and $\Sigma = [\#o_0 \mapsto L, \#o_1 \mapsto H, \#glob \mapsto L]$, the reading effect of the expression $o_1.p$ must be H, because it leaks information about the domain of o_1 whose level is H. Naturally, when an object literal is evaluated, its structure security level is set to the level of the execution context, because the creation of the object is visible at that level. Finally, when creating a new property in an object o, the monitor checks whether the structure security level of o ($\Sigma(\#o)$) is at least as high as the reading effects of: (1) the expression that evaluates to $\#o$ (σ_0) and (2) the expression that evaluates to the name of the property to create (σ_1). Recall that both σ_0 and σ_1 are at least as high as the level of the execution context. Hence, the monitor does also implicitly require that $\sigma_{pc} \leq \Sigma(\#o)$. To illustrate the need for these constraints, consider the expression $o_0 = \{\}, o_1 = \{\}, h \; ? \; (h = o_0) : (h = o_1), h.p = 0, l = o_1.p$. This program is illegal, as the final value of the low variable l depends on the original value of the high variable h. In fact, since the level of h is not lower than or equal to the structure security level of any of the two objects, the monitor blocks the property creation.

Noninterferent Monitor. We say that a security monitor is noninterferent *iff* it preserves the low-equality relation. Informally, an information flow monitor is noninterferent *iff*, for any program e, whenever an attacker cannot distinguish two labeled memories before executing e, then the attacker is also unable to distinguish the final memories.

Theorem 1 (Non-Interferent Monitor). *For any expression e, memories μ and μ', respectively labeled by $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$, reference r, security levels σ_{pc} and σ, and function β s.t. $\mu, \Gamma, \Sigma \approx_{\beta,\sigma} \mu', \Gamma', \Sigma'$, $r, \sigma_{pc} \vdash \langle \mu, e, \Gamma, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Gamma_f, \Sigma_f, \sigma_f \rangle$, and $\beta(r), \sigma_{pc} \vdash \langle \mu', e, \Gamma', \Sigma' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Gamma'_f, \Sigma'_f, \sigma'_f \rangle$; then, there exists a function β' extending β s.t.: $\mu_f, \Gamma_f, \Sigma_f \approx_{\beta',\sigma} \mu'_f, \Gamma'_f, \Sigma'_f$. Moreover, if either $\sigma_f \leq \sigma$ or $\sigma'_f \leq \sigma$, then $v_f \sim_{\beta'} v'_f$.*

3 Monitor-Inlining

This section presents a new information flow monitor-inlining compiler for Core JavaScript, which instruments programs in order to simulate their execution in the monitored semantics presented in Section 2. This instrumentation rests on a technique that consists in pairing up each variable/property with a new one, called its *shadow* variable/property [7,11], 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).

In the design of the compiler, we assume the existence of a given a set of variable and property names, denoted by $\mathcal{I}_C$, that do not overlap with those available for the programmer. In particular, the compilation of every *indexed expression* requires extra variables intended to store the corresponding value and security level, to be later used in the compilation of other expressions that include it. Hence, we assume the set of compiler variables to include two indexed sets of variables $\{\$\hat{l}_i\}_{i\in\mathbf{N}}$ and $\{\$\hat{v}_i\}_{i\in\mathbf{N}}$, used to store the levels and the values of intermediate expressions, respectively. Given a variable x, we denote by $\$l_x$ the corresponding shadow variable. In contrast to variables, whose names are available at compile time, property names are dynamically computed. Therefore, we assume the existence of a runtime function $\$\mathit{shadow}$ that given a property name outputs the name of the corresponding shadow property. Given an expression e to compile, the compiler guarantees that e does not use variable and property names in $\mathcal{I}_C$ by (1) statically verifying that the variables in e do not overlap with $\mathcal{I}_C$ and (2) dynamically verifying that e does not look-up/create/update properties whose names belong to $\mathcal{I}_C$. To this end, the compiler makes use of a runtime function $\$\mathit{legal}$ that returns *true* when its argument does not belong to $\mathcal{I}_C$. For clarity, all identifiers reserved for the compiler are prefixed with a \$. By making sure that compiler identifiers do not overlap with those of the programs to compile, we guarantee the soundness of the proposed transformation even when it receives as input *malicious programs*. Malicious programs try to bypass the inlined runtime enforcement mechanism by rewriting some of its internal variables/properties. E.g., the compilation of the expression $\$\hat{l}_h = L,\ l = h$ fails, as this program tries to tamper with the internal state of the runtime enforcement mechanism in order to be allowed to leak confidential information. Concretely, this program tries to transfer the content of h to l without raising the level of l by setting the level associated with variable h to low.

Besides adding to every object o an additional shadow property $\$l_p$ for every property p in its domain, the inlined monitoring code also adds to o a special property $\$\mathit{struct}$ that stores its structure security level. Hence, 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 $\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$\mathit{struct} \mapsto L]$.

$$\textsc{Value}\quad r,\sigma_{pc}\vdash\langle\mu,v,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu,v,\Gamma,\Sigma,\sigma_{pc}\rangle$$

$$\textsc{This}\quad\frac{r_{this}=\mu(r)(@this)\qquad\sigma_{this}=\Gamma(r)(@this)\sqcup\sigma_{pc}}{r,\sigma_{pc}\vdash\langle\mu,\mathsf{this},\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu,r_{this},\Gamma,\Sigma,\sigma_{this}\rangle}$$

$$\textsc{Binary Operation}\quad\frac{r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,v_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,v_1,\Gamma_1,\Sigma_1,\sigma_1\rangle}{r,\sigma_{pc}\vdash\langle\mu,e_0\ \mathsf{op}\ e_1,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_1,v_0\ \mathsf{op}\ v_1,\Gamma_1,\Sigma_1,\sigma_0\sqcup\sigma_1\rangle}$$

$$\textsc{Variable}\quad\frac{\langle\mu,r,x\rangle\ \mathcal{R}_{Scope}\ r_x\qquad r_x\neq null\qquad v=\mu(r_x)(x)\qquad\sigma=\Gamma(r_x)(x)\sqcup\sigma_{pc}}{r,\sigma_{pc}\vdash\langle\mu,x,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu,v,\Gamma,\Sigma,\sigma\rangle}$$

$$\textsc{Variable Assignment}\quad\frac{r,\sigma_{pc}\vdash\langle\mu,e,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,v_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad\langle\mu_0,r,x\rangle\ \mathcal{R}_{Scope}\ r_x\qquad r_x\neq null\qquad\sigma_{pc}\leq\Gamma_0(r_x)(x)\qquad\Gamma'=\Gamma_0\left[r_x\mapsto\Gamma_0(r_x)\left[x\mapsto\sigma_0\right]\right]\qquad\mu'=\mu_0\left[r_x\mapsto\mu_0(r_x)\left[x\mapsto v_0\right]\right]}{r,\sigma_{pc}\vdash\langle\mu,x=e,\Gamma\rangle\Downarrow_{IF}\langle\mu',v_0,\Gamma',\Sigma_0,\sigma_0\rangle}$$

$$\textsc{Property Look-up}\quad\frac{r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,r_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,m_1,\Gamma_1,\Sigma_1,\sigma_1\rangle\qquad\langle\mu_1,r_0,m_1,\Gamma_1,\Sigma_1\rangle\ \mathcal{R}_{Proto}\ \langle r',\sigma'\rangle\qquad\langle v,\sigma\rangle=\begin{cases}\langle\mu_1(r')(m_1),\sigma_0\sqcup\sigma_1\sqcup\sigma'\sqcup\Gamma_1(r')(m_1)\rangle & \text{if } r'\neq null\\ \langle undefined,\sigma_0\sqcup\sigma_1\sqcup\sigma'\rangle & \text{otherwise}\end{cases}}{r,\sigma_{pc}\vdash\langle\mu,e_0[e_1],\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_1,v,\Gamma_1,\Sigma_1,\sigma\rangle}$$

$$\textsc{Property Assignment}\quad\frac{\begin{array}{c}r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,r_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,m_1,\Gamma_1,\Sigma_1,\sigma_1\rangle\\ r,\sigma_{pc}\vdash\langle\mu_1,e_2,\Gamma_1,\Sigma_1\rangle\Downarrow_{IF}\langle\mu_2,v_2,\Gamma_2,\Sigma_2,\sigma_2\rangle\qquad\Gamma'=\Gamma_2\left[r_0\mapsto\Gamma_2(r_0)\left[m_1\mapsto\sigma_0\sqcup\sigma_1\sqcup\sigma_2\right]\right]\\ \mu'=\mu_2\left[r_0\mapsto\mu_2(r_0)\left[m_1\mapsto v_2\right]\right]\qquad m_1\in\mu_2(r_0)\Rightarrow\sigma_0\sqcup\sigma_1\leq\Gamma_2(r_0)(m_1)\qquad m_1\notin\mu_2(r_0)\Rightarrow\sigma_0\sqcup\sigma_1\leq\Sigma_2(r_0)\end{array}}{r,\sigma_{pc}\vdash\langle\mu,e_0[e_1]=e_2,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',v_2,\Gamma',\Sigma_2,\sigma_2\rangle}$$

$$\textsc{Function Literal}\quad\frac{r_f\notin dom(\mu)\qquad\mu'=\mu\left[r_f\mapsto\left[@fscope\mapsto r,@code\mapsto\lambda x.e\right]\right]\qquad\Gamma'=\Gamma\left[r_f\mapsto\left[@fscope\mapsto\sigma_{pc},@code\mapsto\sigma_{pc}\right]\right]\qquad\Sigma'=\Sigma\left[r_f\mapsto\sigma_{pc}\right]}{r,\sigma_{pc}\vdash\langle\mu,\mathsf{function}(x)\{e\},\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',r_f,\Gamma',\Sigma',\sigma_{pc}\rangle}$$

$$\textsc{Object Literal}\quad\frac{r_o\notin dom(\mu)\qquad\mu'=\mu\left[r_o\mapsto\left[\_prot\_\mapsto null\right]\right]\qquad\Gamma'=\Gamma\left[r_o\mapsto\left[\_prot\_\mapsto\sigma_{pc}\right]\right]\qquad\Sigma'=\Sigma\left[r_o\mapsto\sigma_{pc}\right]}{r,\sigma_{pc}\vdash\langle\mu,\{\},\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',r_o,\Gamma',\Sigma',\sigma_{pc}\rangle}$$

$$\textsc{Function Call}\quad\frac{\begin{array}{c}r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,r_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,v_1,\Gamma_1,\Sigma_1,\sigma_1\rangle\\ \langle\mu_1,\Gamma_1,\Sigma_1,r_0,v_1,\#glob,\sigma_0,\sigma_1\rangle\ \mathcal{R}_{NewScope}\ \langle\hat{\mu},\hat{e},\hat{\Gamma},\hat{\Sigma},\hat{r},\hat{\sigma}_{pc}\rangle\qquad\hat{r},\hat{\sigma}_{pc}\vdash\langle\hat{\mu},\hat{e},\hat{\Gamma},\hat{\Sigma}\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle\end{array}}{r,\sigma_{pc}\vdash\langle\mu,e_0(e_1),\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle}$$

$$\textsc{Method Call}\quad\frac{\begin{array}{c}r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,r_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,m_1,\Gamma_1,\Sigma_1,\sigma_1\rangle\\ r,\sigma_{pc}\vdash\langle\mu_1,e_2,\Gamma_1,\Sigma_1\rangle\Downarrow_{IF}\langle\mu_2,v_2,\Gamma_2,\Sigma_2,\sigma_2\rangle\qquad\langle\mu_2,r_0,m_1,\Gamma_2,\Sigma_2\rangle\ \mathcal{R}_{Proto}\ \langle r_m,\sigma_m\rangle\qquad r_f=\mu_2(r_m)(m_1)\\ \langle\mu_2,\Gamma_2,\Sigma_2,r_f,v_2,r_0,\sigma_0\sqcup\sigma_1\sqcup\Gamma_2(r_m)(m_1)\sqcup\sigma_m,\sigma_2\rangle\ \mathcal{R}_{NewScope}\ \langle\hat{\mu},\hat{e},\hat{\Gamma},\hat{\Sigma},\hat{r},\hat{\sigma}_{pc}\rangle\qquad\hat{r},\hat{\sigma}_{pc}\vdash\langle\hat{\mu},\hat{e},\hat{\Gamma},\hat{\Sigma}\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle\end{array}}{r,\sigma_{pc}\vdash\langle\mu,e_0[e_1](e_2),\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle}$$

$$\textsc{Sequence}\quad\frac{r,\sigma_{pc}\vdash\langle\mu,e_0,\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_0,v_0,\Gamma_0,\Sigma_0,\sigma_0\rangle\qquad r,\sigma_{pc}\vdash\langle\mu_0,e_1,\Gamma_0,\Sigma_0\rangle\Downarrow_{IF}\langle\mu_1,v_1,\Gamma_1,\Sigma_1,\sigma_1\rangle}{r,\sigma_{pc}\vdash\langle\mu,(e_0,e_1),\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu_2,v_1,\Gamma_1,\Sigma_1,\sigma_1\rangle}$$

$$\textsc{Conditional}\quad\frac{r,\sigma_{pc}\vdash\langle\mu,\hat{e},\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\hat{\mu},\hat{v},\hat{\Gamma},\hat{\Sigma},\hat{\sigma}\rangle\qquad i=\begin{cases}0\text{ if }\hat{v}\notin\{0,false,undefined,null\}\\ 1\text{ otherwise}\end{cases}\qquad r,\sigma_{pc}\sqcup\hat{\sigma}\vdash\langle\hat{\mu},e_i,\hat{\Gamma},\hat{\Sigma}\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle}{r,\sigma_{pc}\vdash\langle\mu,\hat{e}\ ?\ (e_0):(e_1),\Gamma,\Sigma\rangle\Downarrow_{IF}\langle\mu',v,\Gamma',\Sigma',\sigma\rangle}$$

$$\textsc{New Scope}\quad\frac{\begin{array}{c}r=\mu(r_f)(@fscope)\qquad\lambda x.\{\mathsf{var}\ y_1,\cdots,y_n;\ e\}=\mu(r_f)(@code)\qquad r'\notin dom(\mu)\qquad\sigma'_{pc}=\sigma_{pc}\sqcup\Gamma(r_f)(@fscope)\\ \mu'=\mu\left[r'\mapsto\left[@scope\mapsto r,x\mapsto v_{arg},y_1\mapsto undefined,\cdots,y_n\mapsto undefined,@this\mapsto r_{this}\right]\right]\qquad\Sigma'=\Sigma\left[r'\mapsto\sigma'_{pc}\right]\\ \Gamma'=\Gamma\left[r'\mapsto\left[@scope\mapsto\sigma'_{pc},x\mapsto\sigma'_{pc}\sqcup\sigma_{arg},y_1\mapsto\sigma'_{pc},\cdots,y_n\mapsto\sigma'_{pc},@this\mapsto\sigma'_{pc}\right]\right]\end{array}}{\langle\mu,\Gamma,\Sigma,r_f,v_{arg},r_{this},\sigma_{pc},\sigma_{arg}\rangle\ \mathcal{R}_{NewScope}\ \langle\mu',e,\Gamma',\Sigma',r',\sigma'_{pc}\rangle}$$

Fig. 2. Monitored Core JavaScript Semantics

Formal Specification. The inlining compiler is defined as a function $\mathcal{C}$, given in Figure 3, that expects as input an expression e and produces a tuple $\langle \hat{e}, i \rangle$, where $\hat{e}$ is the expression that simulates the execution of e in the monitored semantics and i an index such that, after the execution of $\hat{e}$, $\$\hat{v}_i$ stores the value to which e evaluates in the monitored semantics and $\$\hat{l}_i$ its corresponding reading effect. Besides the runtime functions $\$shadow$ and $\$legal$, the compiler makes use of (1) a runtime function $\$check$ that diverges when its argument is different from *true*, (2) a runtime function $\$inspect$ that expects as input an object and a property and outputs the level associated with the corresponding prototype-chain inspection procedure, and (3) an additional binary operator hasOwnProp that checks whether the object given as its left operand defines the property given as its right one. During the evaluation of the instrumented code, the level of the execution context, σ_{pc}, is assumed to be stored in a variable $\$pc$. To this end, function literals are instrumented in order to receive as input the level of the argument and the level of the context in which they are invoked. Function/method calls are instrumented accordingly. Furthermore, the instrumented code of a function/method call must have access to both the return value of the original function/method and the level that is to be associated with that value. Therefore, every function literal returns an object that defines two properties: (1) a property $\$v$ where it stores the return value of the original function and (2) a property $\$l$ where it stores the level to be associated with that value. Each compiler rule precisely mimics the corresponding monitor rule. However, the compiler must also keep track of the variables in which the security level and the value of the expression to compile are stored during execution. This is done by assigning the value to which the expression evaluates to a new variable $\$\hat{v}_i$ and the security level to a new variable $\$\hat{l}_i$. The compilation of every variable/property assignment and sequence expression does not introduce additional variables because the corresponding value and reading effect are already available in the indexed variables introduced by the corresponding subexpressions.

Correctness. Definition 2 presents a *similarity relation* between labeled memories in the monitored semantics and instrumented memories in the original semantics, denoted by $\mathcal{S}_\beta$. $\mathcal{S}_\beta$ requires that for every object in the labeled memory, the corresponding labeling coincide with the instrumented labeling (except for some internal properties whose levels can be automatically inferred) 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 the instrumented object to be larger than the one of the original object.

Definition 2 (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 $p \in dom(o) \backslash \{@scope, @this, @code\}$, $o(p) \sim_{\mathcal{C}(\beta)} o'(p)$ and $\Gamma(r)(p) = o'(\$l_p)$*

$$\textsc{Value}\quad \frac{\hat{e} = \$\hat{l}_i = \$pc,\ \$\hat{v}_i = v}{\mathcal{C}\langle v^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Variable}\quad \frac{x \notin \mathcal{I}_C \qquad \hat{e} = \$\hat{l}_i = \$pc \sqcup \$l_x,\ \$\hat{v}_i = x}{\mathcal{C}\langle x^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{This}\quad \frac{\hat{e} = \$\hat{l}_i = \$pc,\ \$\hat{v}_i = \mathsf{this}}{\mathcal{C}\langle \mathsf{this}^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Binary Operation}\quad \frac{\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, j\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, k\rangle \qquad \hat{e} = \hat{e}_0,\ \hat{e}_1,\ \$\hat{l}_i = \$\hat{l}_j \sqcup \$\hat{l}_k,\ \$\hat{v}_i = \$\hat{v}_j\ \mathsf{op}\ \$\hat{v}_k}{\mathcal{C}\langle e_0\ \mathsf{op}^i\ e_1\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Variable Assignment}\quad \frac{x \notin \mathcal{I}_C \qquad \mathcal{C}\langle e\rangle = \langle e', i\rangle \qquad \hat{e} = e',\ \$check(\$pc \le \$l_x),\ \$l_x = \$\hat{l}_i,\ x = \$\hat{v}_i}{\mathcal{C}\langle x = e\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Property Look-up}\quad \frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, k\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, j\rangle \qquad e_{lev} = \$\hat{l}_i = \$\hat{l}_k \sqcup \$\hat{l}_j \sqcup \$inspect(\$\hat{v}_k, \$\hat{v}_j)\\ \hat{e} = \hat{e}_0,\ \hat{e}_1,\ \$check(\$legal(\$\hat{v}_j)),\ e_{lev},\ \$\hat{v}_i = \$\hat{v}_k[\$\hat{v}_j]\end{array}}{\mathcal{C}\langle e_0[e_1]^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Property Assignment}\quad \frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, i\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, j\rangle \qquad \mathcal{C}\langle e_2\rangle = \langle \hat{e}_2, k\rangle\\ e_{enf} = \$\hat{v}_i\ \mathsf{hasOwnProp}\ \$\hat{v}_j\ ?\ \Big(\$check(\$\hat{l}_i \sqcup \$\hat{l}_j \le \$\hat{v}_i[\$shadow(\$\hat{v}_j)])\Big) : \Big(\$check(\$\hat{l}_i \sqcup \$\hat{l}_j \le \$\hat{v}_i.\$struct)\Big)\\ \hat{e} = \hat{e}_0,\ \hat{e}_1,\ \hat{e}_2,\ \$check(\$legal(\$\hat{v}_j)),\ e_{enf},\ \$\hat{v}_i[\$shadow(\$\hat{v}_j)] = \$\hat{l}_i \sqcup \$\hat{l}_j \sqcup \$\hat{l}_k, \$\hat{v}_i[\$\hat{v}_j] = \$\hat{v}_k\end{array}}{\mathcal{C}\langle e_0[e_1] = e_2\rangle = \langle \hat{e}, k\rangle}$$

$$\textsc{Function Literal}\quad \frac{\begin{array}{c}\mathcal{C}\langle e\rangle = \langle \hat{e}_f, j\rangle \qquad e_{fbody} = \hat{e}_f,\ \$ret = \{\},\ \$ret.\$v = \$\hat{v}_j,\ \$ret.\$l = \$\hat{l}_j,\ \$ret\\ \{i_1, \cdots, i_k\} = indexes(e) \qquad e_f = \$\hat{v}_i = \mathsf{function}(x, \$l_x, \$pc)\{\mathsf{var}\ y_1, \cdots, y_n, \$\hat{v}_{i_1}, \$\hat{l}_{i_1}, \cdots, \$\hat{v}_{i_k}, \$\hat{l}_{i_k};\ e_{fbody}\}\\ \hat{e} = e_f,\ \$\hat{v}_i.\$struct = \$pc,\ \$\hat{v}_i.\$l_{@fscope} = \$pc,\ \$\hat{l}_i = \$pc,\ \$\hat{v}_i\end{array}}{\mathcal{C}\langle \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Object Literal}\quad \frac{e' = \$\hat{v}_i.\$struct = \$pc,\ \$\hat{v}_i.\$l_{proto} = \$pc \qquad \hat{e} = \$\hat{v}_i = \{\},\ e',\ \$\hat{l}_i = \$pc,\ \$\hat{v}_i}{\mathcal{C}\langle\{\}^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Function Call}\quad \frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, j\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, k\rangle\\ e' = \$\hat{l}_{ctx} = \$\hat{v}_j.\$l_{@fscope} \sqcup \$\hat{l}_j,\ \$ret = \$\hat{v}_j(\$\hat{v}_k, \$\hat{l}_k \sqcup \$\hat{l}_{ctx}, \$\hat{l}_{ctx})\\ \hat{e} = \hat{e}_0,\ \hat{e}_1,\ e',\ \$\hat{l}_i = \$ret.\$l,\ \$\hat{v}_i = \$ret.\$v\end{array}}{\mathcal{C}\langle e_0(e_1)^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Method Call}\quad \frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, j\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, k\rangle \qquad \mathcal{C}\langle e_2\rangle = \langle \hat{e}_2, l\rangle\\ e' = \hat{e}_0,\ \hat{e}_1,\ \hat{e}_2,\ \$\hat{l}_{ctx} = \$\hat{l}_j \sqcup \$\hat{l}_k \sqcup \$inspect(\$\hat{v}_k, \$\hat{v}_j) \sqcup \$\hat{v}_j[\$\hat{v}_k].\$l_{@fscope}\\ \hat{e} = e',\ \$ret = \$\hat{v}_j[\$\hat{v}_k](\$\hat{v}_l, \$\hat{l}_{ctx} \sqcup \$\hat{l}_l, \$\hat{l}_{ctx}),\ \$\hat{l}_i = \$ret.\$l,\ \$\hat{v}_i = \$ret.\$v\end{array}}{\mathcal{C}\langle e_0[e_1](e_2)^i\rangle = \langle \hat{e}, i\rangle}$$

$$\textsc{Sequence}\quad \frac{\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, i\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, j\rangle \qquad \hat{e} = \hat{e}_0, \hat{e}_1}{\mathcal{C}\langle e_0, e_1\rangle = \langle \hat{e}, j\rangle}$$

$$\textsc{Conditional}\quad \frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0, i\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1, j\rangle \qquad \mathcal{C}\langle e_2\rangle = \langle \hat{e}_2, k\rangle\\ e_{cond} = \$\hat{v}_i\ ?\ \Big(\hat{e}_1,\ \$\hat{v}_t = \$\hat{v}_j,\ \$\hat{l}_t = \$\hat{l}_j\Big) : \Big(\hat{e}_2,\ \$\hat{v}_t = \$\hat{v}_k,\ \$\hat{l}_t = \$\hat{l}_k\Big)\\ \hat{e} = \hat{e}_0,\ \$\hat{l}_s = \$pc,\ \$pc = \$pc \sqcup \$\hat{l}_i,\ e_{cond},\ \$pc = \$\hat{l}_s, \$\hat{v}_t\end{array}}{\mathcal{C}\langle e_0\ ?^{s,t}\ (e_1) : (e_2)\rangle = \langle \hat{e}, t\rangle}$$

Fig. 3. Information Flow Monitor Inlining Compiler

The Correctness Theorem states that, provided that a program and its compiled counterpart are evaluated in similar configurations, 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 configurations as well as the computed values are similar. Therefore, since the monitored semantics only allows secure executions to go through, we guarantee that, when using the inlining compiler, programs are rewritten in such a way that only their secure executions are allowed to terminate.

Theorem 2 (Correctness). *Provided that e does not use identifiers in $\mathcal{I}_C$, for any labeled and instrumented configurations $\langle \mu, e, \Gamma, \Sigma\rangle$ and $\langle \mu', e'\rangle$, function β, and reference r in μ, such that $\langle \mu, \Gamma, \Sigma\rangle \; \mathcal{S}_\beta \; \mu'$ and $\mathcal{C}\langle e\rangle = \langle e', i\rangle$, for some index i; there exists $\langle \mu_f, v_f, \Gamma_f, \sigma\rangle$ such that $r, \bot \vdash \langle \mu, e, \Gamma, \Sigma\rangle \Downarrow_{IF} \langle \mu_f, v_f, \Gamma_f, \Sigma_f, \sigma_f\rangle$ iff there exists $\langle \mu'_f, v'_f\rangle$ such that $\beta(r) \vdash \langle\langle \mu', e'\rangle\rangle \Downarrow \langle\langle \mu'_f, v'_f\rangle\rangle$, 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) $\sigma_f = \mu'_f(\beta(r))(\$\hat{l}_i)$.*

4 Discussion and Related Work

JavaScript Semantics. Since scope objects are assumed not to have a prototype and since we do not include the JavaScript `with` construct, Core JavaScript programs are syntactically scoped. This means that we could have modeled the binding of variables using substitution, as in other works targeting subsets of the whole language, as [8]. However, we have chosen to model scope using scope objects, as in [10], for two main reasons. First, we envisage to extend the model to deal with a larger subset of the language. Second, by modeling the binding of variables in the same way we model the binding of properties, we do not need to introduce an extra labeling function for the labeling of variables.

Monitoring Secure Information Flow. Information flow monitors can be divided in two main classes. *Purely dynamic* monitors (such as [3] and [4]) do not make use of any kind of static analysis. On the contrary, *hybrid monitors* (such as [12]) make use of static analyses to reason about the implicit flows that can arise due to untaken execution paths. Our choice for the inlining of a purely dynamic monitor has to do with the fact that the dynamic features of JavaScript make it very difficult to approximate the resources created/updated in untaken program branches. Hedin and Sabelfeld [9] have been the first to design an information flow monitor for a realistic core of JavaScript. Their monitor is purely dynamic and enforces the no-sensitive-upgrade discipline. This monitor has been designed in order to guide a browser instrumentation and not an inlining transformation. Furthermore, it differs from ours in that it labels values instead of variables/properties. Bichhawat et al. [6] have recently proposed a hybrid monitor that makes use of a sophisticated static analysis to minimize performance overhead [6].

Monitor-Inlining Transformations. Chudnov and Naumann [7] propose an information flow monitor inlining transformation for a WHILE language, which inlines the hybrid information flow monitor presented in [12]. Simultaneously,

Magazinius et al. [11] propose the inlining of a purely dynamic information flow monitor that enforces the no-sensitive-upgrade discipline for a simple imperative language that features global functions, a let construct, and an *eval* expression that allows for dynamic code evaluation. Both compilers pair up each variable with a *shadow* variable. We extend this technique to handle object properties by pairing up each property with a shadow property. The languages modeled in both [7] and [11] only feature primitive values and do not feature scope composition (in [7] there are no functions and in [11] every function is executed in a "clean" environment and does not produce side-effects). Hence, in both [7] and [11], the reading effect of an expression e corresponds to the least upper bound on the levels of the variables of e. Therefore, the instrumented code for computing the level of e is simply $\$l_{x_1} \sqcup \cdots \sqcup \$l_{x_n}$, where $\{x_1, \cdots, x_n\}$ are the variables that explicitly occur in e. In Core JavaScript (as in JavaScript) this does not hold. First, one can immediately see that expressions that feature property look-ups or function/method calls do not generally verify this property. Second, expressions may be composed of expressions that have side effects. Therefore, the level associated with the whole expression can actually be lower than the least upper bound on the levels of the variables that it includes. As an example, consider the expression $(x = y) + x$. Since $x = y$ evaluates to the value of y (besides assigning the value of y to x), the level of the whole expression only depends on the initial level of y. In order to handle these two issues, the inlining transformation must introduce extra variables to keep track of the values and levels of intermediate expressions. Finally, both [7] and [11] ignore the problem of malicious programs.

In summary, we have presented the first compiler for securing information flow in an important subset of JavaScript. The presented compiler is proven sound even when it is given as input malicious code that actively tries to bypass the inlined enforcement mechanism. A prototype of the compiler is available via [1] together with a broad set of examples that illustrate its applicability.

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

References

1. Information flow monitor-inlining compiler, `http://www-sop.inria.fr/members/Jose.Santos/`
2. The 5th edition of ECMA 262 June 2011. ECMAScript Language Specification. Technical report, ECMA (2011)
3. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. PLAS (2009)
4. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. PLAS (2010)
5. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: CSFW (2002)
6. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in WebKit's JavaScript bytecode. In: Abadi, M., Kremer, S. (eds.) POST 2014 (ETAPS 2014). LNCS, vol. 8414, pp. 159–178. Springer, Heidelberg (2014)

7. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: CSF (2010)
8. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010)
9. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: CSF (2012)
10. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
11. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly inlining of dynamic security monitors. In: Computers & Security (2012)
12. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: CSF (2010)
13. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications (2003)