A Monitor Inlining Compiler for Securing JavaScript Programs

José Fragoso Santos and Tamara Rezk

INRIA
firstname.lastname@inria.fr

Abstract. JavaScript is a popular language used to program client-side web applications. Such applications can include untrusted code dynamically loaded from third party code providers (such as online advertisements). This issue raises the need for enforcement mechanisms to ensure security properties for JavaScript programs. The dynamic nature of the JavaScript programming language makes it a hard target for static analysis. Hence, research on mechanisms for enforcing security properties for JavaScript programs has mostly focused on dynamic approaches, such as runtime monitoring and program instrumentation. We design and implement a novel compiler that inlines a security monitor and we formally prove it correct with respect to an information flow security property. To the best of our knowledge, it is the first proven correct information flow monitor inlining transformation for JavaScript programs.

1 Introduction

The growing impact of web applications on sensitive aspects of daily life (e-banking, e-voting, e-commerce) has brought the problem of web security to the public scene. There is an unprecedented need for practical mechanisms to protect applications against security violations. A recent study [13] shows that 50000 JavaScript-based websites, including Alexa global top-100 websites, present privacy violating information flows vulnerabilities. Violations [7] on the client side of web applications are mainly due to the permissive JavaScript semantics and integration of code coming from different origins, such as advertisement scripts. This critical security situation has led to an increasing interest in sound and practical mechanisms to enforce JavaScript secure information flow.

Due to the dynamic nature of JavaScript, recent works on JavaScript security focus on dynamic mechanisms such as security monitoring [12], secure multi-execution [10], and multi-facets [5]. There are two main approaches for implementing a JavaScript security monitor: either modify a JavaScript engine so that it also implements the security monitor (as in [12]), or inline the monitor in the original program (as in [8,16]). The second approach is *browser-independent* and can be deployed immediately in practice. On the contrary, because of the diversity of browsers available and the impossibility of predicting where the application will execute, a security solution that requires browser modifications can often be ineffective.

In this work we present a compiler that inlines an information flow monitor for JavaScript. The compiler is browser-independent and its purpose is to protect client-side web applications from malicious or unintended information leaks. In order to instrument JavaScript programs in an efficient manner, we consider a new information flow monitor and prove it correct. We formally prove our compiler correct with respect to the monitored semantics. To the best of our knowledge, it is the first proven correct information flow monitor inlining transformation for JavaScript programs. The implementation of our compiler takes into account: special features of JavaScript semantics such as implicit type coercions and programs that actively try to bypass the inlined enforcement mechanisms. Specifically, we guarantee that third-party programs cannot (1) access the compiler internal state by randomizing the names of the resources through which it is accessed and (2) change the behaviour of native functions that are used by the enforcement mechanisms inlined in the compiled code.

Limitations. We consider a core fragment of JavaScript as described in the ECMA Standard 2009 [2]. In particular, from the standard, we do not implement the following constructs: switch, with, break/continue, labeled statements, getters and setters, and try/catch/finally statement.

Related work. In web applications there is a need for dynamic mechanisms (see LeGuernic thesis for a survey on the subject [14]) for information flow control due to the dynamic nature of the JavaScript language. Austin and Flanagan [5] and Hedin and Sabelfeld [12] study and implement runtime monitors for enforcing noninterference in JavaScript. The monitor we propose has substantially simpler constraints than that of Hedin and Sabelfeld. Besides we label properties instead of values which makes the inlining process more direct.

Venkatakrishnan [17] is the first to present a hybrid technique that relies on runtime information-flow tracking augmented with static analysis to reason about implicit flows that arise due to unexecuted paths in a program. Chudnov and Naumann [8] propose the inlining of an information flow monitor for a simple imperative language. In parallel, Magazinius et al. [15, 16] also propose a monitor inlining transformation for an imperative language with the novel feature of performing inlining on the fly to handle *eval*. In our implementation we apply the techniques presented in [8, 16] and adapt them to the JavaScript language.

2 A JavaScript Information Flow Monitor

JavaScript Syntax and Semantics The syntax of the JavaScript subset formally considered in this work is given in Figure 1. Objects are a fundamental datatype in JavaScript. Informally, one can say that an object is a collection of named values. Formally, at the semantic level, we model objects as partial functions from strings, taken from a set $\mathcal{S}tr$, to values, taken from $\mathcal{P}rim \cup \mathcal{R}ef$, where $\mathcal{P}rim$ is the set of primitive values and $\mathcal{R}ef$ the set of references. $\mathcal{P}rim$ includes strings, numbers, and booleans, as well as two special values *null* and *undefined*. Some properties cannot be changed by the program, for clarity those

$e ::=$	x	identifier	$s ::=$	$e;$	expression statement
	v	primitive values		$s_1\ s_2$	sequence of statements
	this	this keyword		$\mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}$	conditional
	$\mathsf{function}(x)\{s\}$	function literal		$\mathsf{while}(e)\{s\}$	while loop
	$e_1[e_2]$	member selector	$v ::=$	n	number
	$x = e$	variable assignment		m	string
	$e_1[e_2] = e_3$	property assignment		b	boolean
	$\mathsf{new}\ e_1(e_2)$	constructor call		*undefined*	undefined
	$e_1(e_2)$	function call		*null*	null
	$e_1[e_2](e_3)$	method call		r	reference
	$(e_1\ \mathsf{op}_2\ e_2)$	binary operations		$\lambda x.s$	function runtime value

Fig. 1. JavaScript Core Syntax

properties are prefixed with "@". Finally, a memory μ is a mapping from references to objects. Every expression that creates an object in memory yields a reference that points to it, which is non-deterministically chosen. Hence, references can be viewed as pointers to objects. Interestingly, the evaluation of a function literal also yields a reference to an object, deemed a *function object*, that represents that function literal. Particularly, since functions are executed in the environment on which they are defined, function objects must include the code of the corresponding function (stored in property @*code*) and a reference to a special object that captures the environment that was active when the corresponding function literal was evaluated (stored in property *@fscope*). These special objects are called scope objects. A scope object is an object that maps the formal argument of the function that is executing to its current value and property @*scope* to the reference of the scope object that was active when the function that is executing was stored in memory. Additionally, every function object defines a property *prototype*. The prototype of an object created by calling a function f as a constructor is f.*prototype*. Upon the invocation of a function f, the JavaScript interpreter starts by creating a new scope object that maps the formal parameter of f, as well as the variables declared in its body, to their respective values. Moreover it maps property @*scope* to f.*@fscope*, which is the reference of the scope object that was active when the function literal associated with f was evaluated.

Example 1 (Scope Objects). Consider the following program:

```
1 var output;
2 var f = function(x) {
3    var g, h;
4    g = function (x) { h();}
5    h = function () { output = x; }
6    g(0);
7 }
8 f(1);
```

Its execution creates the following scope objects: $o_s^f = [x \mapsto 1, @scope \mapsto \#global]$, $o_s^g = [x \mapsto 0, @scope \mapsto \#o_s^f]$, and $o_s^h = [@scope \mapsto \#o_s^f]$, where $\#o_s^f$ denotes the reference that points to o_s^f.

The sequence of scope objects that can be accessed from a given scope object through the respective $@scope$ properties is deemed a *scope chain*. In order to determine the value associated with a given variable, one has to inspect all the objects in the scope chain starting from the *active* scope object (that is, the one at the top of the scope chain). This behaviour is modeled by the semantic relation $\mathcal{R}_{Scope}$ which is presented in Definition 1. If $\langle \mu, r_1, m \rangle \; \mathcal{R}_{Scope} \; r_2$, then r_2 points to the scope object that is closest to the one pointed by r_1 in the corresponding scope chain (that is stored in memory μ) and which defines a binding for m. The *global object*, which is assumed to be stored in a fixed reference $\#global$ and which defines the bindings for global variables, is the bottom of every scope chain.

Definition 1 ($\mathcal{R}_{Scope}$)**.** *$\mathcal{R}_{Scope}$ is recursively defined as follows:*

$$\textsc{Null}\quad \langle \mu, null, m \rangle \; \mathcal{R}_{Scope} \; null \qquad\qquad \textsc{Base}\quad \frac{m \in \mathsf{dom}(\mu(r))}{\langle \mu, r, m \rangle \; \mathcal{R}_{Scope} \; r}$$

$$\textsc{Look-up}\quad \frac{m \notin \mathsf{dom}(\mu(r)) \qquad \langle \mu, \mu(r, @scope), m \rangle \; \mathcal{R}_{Scope} \; r'}{\langle \mu, r, m \rangle \; \mathcal{R}_{Scope} \; r'}$$

Example 2 (Scope Look-up). After the execution of the program given in Example 1, the global object maps variable *output* to 1 and not to 0, because the scope object that is closest to o_s^h in its corresponding scope chain and which defines a binding for x is o_s^f and not o_s^g (which does not belong to the scope chain of o_s^h).

Objects are created using the new keyword. The evaluation of the expression new$f$ creates a new object whose prototype coincides with $f.prototype$ and then executes function f in an environment in which the *this* keyword is bound to the newly created object.

Example 3. Consider the program below:

```
1 var Person = function(id, name) {
2   this.name = name; this.id = id;
3 };
4 var a1 = new Animal(); a1.foo = 'bar';
5 Person.prototype = a1;
6 var p1 = new Person(1, 'John');
```

The invocation of *Person* as a constructor in line 6, triggers the creation of a new object whose prototype is the animal object instantiated in line 4. Then, the body of *Person* executes in an environment in which the *this* keyword is bound to the newly created object. Hence, after executing this program the object bound to variable $p1$ defines two properties *id* and *name* respectively bound to 1 and "John" and its prototype is the object bound to variable $a1$.

In our model, every object stores a reference to its prototype in an internal property denoted by @*proto*. When a program tries to access a property m of an object o, JavaScript first checks if o has a property named m (that is, if $m \in \mathsf{dom}(o)$). If it does not, JavaScript checks if the prototype of object o has a property named m and so forth. The sequence of objects that can be accessed from a given object through the respective @*proto* properties is deemed a *prototype chain*. The prototype chain look-up process is emulated by the semantic relation $\mathcal{R}_{Proto}$, presented in Definition 2. If $\langle \mu, r, m, \Gamma\rangle \ \mathcal{R}_{Proto} \ \langle r', \sigma\rangle$, then r' is the closest reference to r in its corresponding prototype chain that defines a binding for m. Definition 2 also includes the reasoning about the security levels associated with looking-up the value of a property, which we introduce in the next section.

Definition 2 ($\mathcal{R}_{Proto}$). *$\mathcal{R}_{Proto}$ is recursively defined as follows:*

$$\textsc{Null}\quad \langle \mu, null, m, \Gamma\rangle \ \mathcal{R}_{Proto} \ \langle null, \bot\rangle \qquad \textsc{Base}\quad \frac{m \in \mathsf{dom}(\mu(r))}{\langle \mu, r, m, \Gamma\rangle \ \mathcal{R}_{Proto} \ \langle r, \Gamma(r, m)\rangle}$$

$$\textsc{Look-up}\quad \frac{m \notin \mathsf{dom}(\mu(r)) \qquad r' \doteq \mu(r, @proto) \qquad \langle \mu, r', m, \Gamma\rangle \ \mathcal{R}_{Proto} \ \langle r'', \sigma\rangle \qquad \sigma' \doteq \Gamma(r, @proto) \sqcup \Gamma(r) \sqcup \sigma}{\langle \mu, r, m, \Gamma\rangle \ \mathcal{R}_{Proto} \ \langle r'', \sigma'\rangle}$$

Example 4 (Prototype Look-up). The evaluation of $p1.foo$ after the execution of the program given in Example 3 yields the string "bar" because, although $p1$ does not define property foo, its prototype $a1$ maps foo to "bar".

Since functions evaluate to references, they can be stored as properties of objects. In JavaScript functions that are stored as properties of an object are deemed its methods. The main difference between calling a function as a method and calling a function "as a function" is that when calling a function as a method the *this* keyword is bound to the corresponding object, in other case it is bound to the global object.

Example 5 (Method call semantics). Suppose that the object $Person.prototype$ of Example 3 defines a method *talk* whose code is $\mathsf{function}()\{alert(this.name)\}$. The evaluation of $p1.talk()$ triggers the display of a window showing "John".

The JavaScript semantics relation we consider here has the following form: $r \vdash \langle \mu, s\rangle \Downarrow_{JS} \langle \mu', v\rangle$ where r is the reference of the active scope object, μ and μ' are the original and the final memories respectively, s the statement to be executed, and v the value to which it evaluates. Due to space constraints we do not present the JavaScript semantics here. Instead, we only present the monitored semantics. Obtaining the semantic rules of $\Downarrow_{JS}$ from $\Downarrow_{IF}$ is done by removing from the rules of $\Downarrow_{IF}$ labeling constrains and monitor constrains. Thus, the following correspondence between the two semantics can be straightforwardly proven:

Lemma 1 (Semantics Correspondence). *If $r_s, pc \vdash \langle \mu, s, \Gamma\rangle \Downarrow_{IF} \langle \mu', v, \Gamma', \sigma\rangle$ then $r_s \vdash \langle \mu, s\rangle \Downarrow_{JS} \langle \mu', v\rangle$.*

Security Monitor 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, resources labeled with H are allowed to depend on resources labeled with L, but not the contrary since that would entail an information leak.

Dynamic Labelings. In JavaScript, resources are dynamically created which makes it infeasible to refer to them precisely at the static level. Hence, instead of considering static labelings, we consider dynamic labelings that map each property in every object to a security level. Since every variable is semantically modeled as a property of a given scope object, dynamic labelings also map variables to security levels. Variables and properties are thus treated uniformly. Formally, given a *security lattice* $\mathcal{L}$, a security labeling is a partial function that maps the elements of $\mathcal{R}ef \times \mathcal{S}tr \cup \mathcal{R}ef$ to $\mathcal{L}$. Hence, given an object o stored in a reference r_o, $\Gamma(r_o, p)$ corresponds to the security level associated with o's property p. The level $\Gamma(r_o)$ corresponds to the *structure security level* of o. The structure security level of an object can be understood as the security level associated with its domain.

In JavaScript we need to associate a security level with the domain of every object because it is possible for a program to leak information *via* the domain of an object. We illustrate this point with the following example.

Example 6 (The need for the struture security level).

```
1 var f = function() {};
2 var o = new f;
3 if (h) {
4    o.p = 0;
5 }
6 l = o.p;
```

After the execution of this program, the value of the low variable l depends on the value of the high variable h. Precisely, when $h = 1$, l is set to 0. When $h = 0$, property p is not added to the object. In JavaScript, when a program tries to read the value of a property that does not exist, it does not get an error. Instead, it gets *undefined*. This feature of the language is the reason why one needs to assign a security level to the domain of an object.

From Example 6 one can conclude that when a program tries to read the value of a property p of an object o, if $p \notin \mathsf{dom}(o)$, the security level associated with this property look-up should be equal or higher than the structure security level of o, because this property look-up leaks information about the domain of o. Hence the need to take into account the structure security levels of all objects that comprise a prototype chain in Definition 2. In this definition, the final level associated with the prototype-chain look-up process takes into consideration: (1) the structure security levels of all the objects that are in the prototype chain but do not define a binding for the searched property and (2) the levels of property @*proto* of the same objects.

Example 7. Recall Example 3. Assume that properties of $p1$ can be added in a secret context, that is $\Gamma(\#p1) = H$, where $\#p1$ is the reference that points to $p1$. Then, looking up a property through $p1$ that is not directly defined in $p1$ must take into account its structure security level. This is reflected in the computation of σ' in Definition 2.

Upgrade Instructions. Although dynamic security labelings are constructed at runtime, the programmer must be able to specify at the static level which security levels are to be assigned to the resources that are created during execution. To this end, we extend the JavaScript syntax presented in Definition 1 with three additional constructs, where x and o are identifiers and p is a string:

- $\mathsf{upgVar}(x, \sigma)$ upgrades the level of variable x to the least upper bound between its current level and σ.
- $\mathsf{upgProp}(o, p, \sigma)$ upgrades the level of property p of the object referenced by variable o to the least upper bound between its current level and σ.
- $\mathsf{upgStruct}(o, \sigma)$ upgrades the structure security level of the object referred by o to the least upper bound between its current level and σ.

Low-Equality on Memories. In order for us to formally claim the soundness of our information flow monitor, we introduce a notion of indistinguishability between memories for an adversary that can observe at security level σ, deemed low-equality for memories, denoted by $\approx_{\beta,\sigma}$, and given in Definition 3. Informally, two labeled memories are low-equal at level σ if they coincide in the resources labeled with levels that are $\leq \sigma$. Observe that since references are non-deterministically chosen we need to be able to relate references in one memory with references in the other. To this end we parameterize the low-equality relation with a partial injective function $\beta : \mathcal{R}ef \hookrightarrow \mathcal{R}ef$ [6] that relates observable references. The low-equality definition relies on a binary relation on values that are visible in each object, named β-equality. *β-Equality:* two objects are β-related if they have the same domain and all their corresponding properties are β-related, primitive values are β-related if equal, function runtime values are β-related if syntactically equal, and two references r_0 and r_1 are β-related if the latter is the image of the former. In the following, if f is a function and V a subset of its domain, let $f|_V$ be the restriction of f to V.

Definition 3 (Low-Equality). *Two memories μ_0 and μ_1 are said to be low equal with respect to Γ_0 and Γ_1, a security level σ, and a partial injective function $\beta : \mathcal{R}ef \hookrightarrow \mathcal{R}ef$, written $\mu_0, \Gamma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1$, if μ_0 and μ_1 are labeled by Γ_0 and Γ_1 respectively, and for all references $r \in \mathsf{dom}(\beta)$, the following holds:*

1. $\{p \in \mathsf{dom}(\mu_0(r)) \mid \Gamma_0(r,p) \leq \sigma\} = \{p \in \mathsf{dom}(\mu_1(\beta(r))) \mid \Gamma_1(\beta(r),p) \leq \sigma\} = P$*; i.e. low domains coincide.*
2. $\Gamma_0|_{r,P} = \Gamma_1|_{\beta(r),P}$*; i.e. the two labelings coincide in the common low domain.*
3. $\mu_0(r)|_P \sim_\beta \mu_1(\beta(r))|_P$*; i.e. the two objects obtained by projecting the original objects onto their respective low domain are related.*

4. $\big(\Gamma_0(r), \Gamma_1(\beta(r)) \leq \sigma \wedge \mathsf{dom}(\mu_0(r)) = \mathsf{dom}(\mu_1(\beta(r)))\big) \vee \Gamma_0(r), \Gamma_1(\beta(r)) \not\leq \sigma$; *i.e. either the structure security level in the two objects i s visible and their domains coincide or the structure level of both objects is not visible.*

Monitored Semantics. The monitored semantics relation, $\Downarrow_{IF}$, has the following form:

$$r_s, pc \vdash \langle \mu, s, \Gamma \rangle \Downarrow_{IF} \langle \mu', v, \Gamma', \sigma \rangle$$

where r_s is a reference to the active scope object, μ and μ' are original and final memories, s is the statement to execute, v the value to which it evaluates, Γ and Γ' are the original and final labelings, and σ the *reading effect* of s (that is, the least upper bound on the levels of the resources that are evaluated during the execution of s). Figures 2 and 3 present the rules that define $\Downarrow_{IF}$. We omit the rules corresponding to the the while statement, the if statement, the sequence statement, and the expression statement because they are standard.

In the following, we use $r, pc \vdash \langle \mu, s, \Gamma \rangle \not\Downarrow_{IF}$ to denote program divergence (Note that in this semantics, divergence means that there is not a final configuration such that a semantics derivation exists). If a constraint of the monitor is not satisfied, we say the execution diverges, because no derivation is possible.

The constraints enforced by the monitor are explained below. Essentially, these constraints are intended to forbid the so called *sensitive upgrades* [4] (visible changes in non-visible contexts). Below we give an example that explains why sensitive upgrades cannot be allowed by the monitor.

Example 8 (Why sensitive upgrades must be forbidden). Suppose that the security monitor allows the execution of the program $\mathsf{if}(h)\{l = 0;\}$ else $\{null;\}$ to go through for an initial memory that maps h to 1, but it raises the level of l to H (that is, it performs a sensitive upgrade). If the 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, meaning that the attacker has learned something about the confidential resources of the program. Concretely, considering a monitor that allows sensitive upgrades, if l is visible after executing this program, the attacker can conclude that $h \in \{null, undefined, 0, false\}$.

We present the complete set of constraints used in $\Downarrow_{IF}$:

- [Property Update] $\sigma_0 \sqcup \sigma_1 \leq \Gamma_2(r_0, m_1)$. Levels σ_0 and σ_1 correspond to the reading effects of expressions evaluated in order to update property m_1. The constraint prevents flows of information from the reading effects visible at the level of m_1. Note that reading effect levels are higher than the level of the context in which they are obtained, hence the constraint also prevents sensitive upgrades.
- [Property Creation] $\sigma_0 \sqcup \sigma_1 \leq \Gamma_2(r_0)$. Levels σ_0 and σ_1 correspond to the reading effects of expressions evaluated in order to create property m_1. The constraint prevents flows from the reading effects to the structure level of the object being extended.

PROPERTY UPDATE

$$\frac{r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_0, e_1, \Gamma_0 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Gamma_1, \sigma_1 \rangle \\ r, pc \vdash \langle \mu_1, e_2, \Gamma_1 \rangle \Downarrow_{IF} \langle \mu_2, v_2, \Gamma_2, \sigma_2 \rangle \qquad m_1 \in \mu_2(r_0) \qquad \sigma_0 \sqcup \sigma_1 \leq \Gamma_2(r_0, m_1) \\ \Gamma' \doteq \Gamma_2\left[(r_0, m_1) \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\right] \qquad \mu' \doteq \mu_2\left[(r_0, m_1) \mapsto v_2\right]}{r, pc \vdash \langle \mu, e_0[e_1] = e_2, \Gamma \rangle \Downarrow_{IF} \langle \mu', v_2, \Gamma', \sigma_2 \rangle}$$

PROPERTY CREATION

$$\frac{r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_0, e_1, \Gamma_0 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Gamma_1, \sigma_1 \rangle \\ r, pc \vdash \langle \mu_1, e_2, \Gamma_1 \rangle \Downarrow_{IF} \langle \mu_2, v_2, \Gamma_2, \sigma_2 \rangle \qquad m_1 \notin \mu_2(r_0) \qquad \sigma_0 \sqcup \sigma_1 \leq \Gamma_2(r_0) \\ \mu' \doteq \mu_2\left[(r_0, m_1) \mapsto v_2\right] \qquad \Gamma' \doteq \Gamma_2\left[(r_0, m_1) \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\right]}{r, pc \vdash \langle \mu, e_0[e_1] = e_2, \Gamma \rangle \Downarrow_{IF} \langle \mu', v_2, \Gamma', \sigma_2 \rangle}$$

PROPERTY LOOK-UP

$$\frac{r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_1, e_1, \Gamma_1 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Gamma_1, \sigma_1 \rangle \qquad \langle \mu_1, r_0, m_1, \Gamma_1 \rangle \; \mathcal{R}_{Proto} \; \langle r', \sigma' \rangle}{r, pc \vdash \langle \mu, e_0[e_1], \Gamma \rangle \Downarrow_{IF} \langle \mu_1, \mu_1(r', m_1), \Gamma_1, \sigma_0 \sqcup \sigma_1 \sqcup \sigma' \rangle}$$

VARIABLE

$$\frac{\langle \mu, r, x \rangle \; \mathcal{R}_{Scope} \; r_x \qquad r_x \neq null}{r, pc \vdash \langle \mu, x, \Gamma \rangle \Downarrow_{IF} \langle \mu, \mu(r_x, x), \Gamma, \Gamma(r_x, x) \sqcup pc \rangle}$$

ASSIGNMENT - 1

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

ASSIGNMENT - 2

$$\frac{r, pc \vdash \langle \mu, e, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Gamma_0, \sigma_0 \rangle \qquad \langle \mu, r, x \rangle \; \mathcal{R}_{Scope} \; null \qquad pc \leq \Gamma_1(\#global) \\ \Gamma' \doteq \Gamma_0\left[(\#global, x) \mapsto \sigma_0\right] \qquad \mu' \doteq \mu_0\left[(\#global, x) \mapsto v_0\right]}{r, pc \vdash \langle \mu, x = e, \Gamma \rangle \Downarrow_{IF} \langle \mu', v_0, \Gamma', \sigma_0 \rangle}$$

THIS

$$\frac{}{r, pc \vdash \langle \mu, this, \Gamma \rangle \Downarrow_{IF} \langle \mu, \mu(r, @this), \Gamma, pc \sqcup \Gamma(r, @this) \rangle}$$

VALUE

$$r, pc \vdash \langle \mu, v, \Gamma \rangle \Downarrow_{IF} \langle \mu, v, \Gamma, pc \rangle$$

BIN OPERATOR

$$\frac{r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu, e_1, \Gamma \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Gamma_1, \sigma_1 \rangle}{r, pc \vdash \langle \mu, e_0 \; \mathsf{op}_2 \; e_1, \Gamma \rangle \Downarrow_{IF} \langle \mu_1, \Downarrow_{\mathsf{op}_2}(v_0, v_1), \Gamma_1, \sigma_0 \sqcup \sigma_1 \rangle}$$

VARIABLE UPGRADE

$$\frac{\langle \mu, r, x \rangle \; \mathcal{R}_{Scope} \; r_x \neq undefined}{r, pc \vdash \langle \mu, \mathsf{upgVar}(x, \sigma), \Gamma \rangle \Downarrow_{IF} \langle \mu, undefined, \Gamma\left[(r_x, x) \mapsto \Gamma(r_x, x) \sqcup \sigma\right], pc \rangle}$$

PROPERTY UPGRADE

$$\frac{\langle \mu, r, o \rangle \; \mathcal{R}_{Scope} \; r' \neq undefined \qquad r_o = \mu(r')(o) \qquad p \in \mathsf{dom}(\mu(r_o))}{r, pc \vdash \langle \mu, \mathsf{upgProp}(o, p, \sigma), \Gamma \rangle \Downarrow_{IF} \langle \mu, undefined, \Gamma\left[(r_o, p) \mapsto \Gamma(r_o, p) \sqcup \sigma\right], pc \rangle}$$

STRUCTURE UPGRADE

$$\frac{\langle \mu, r, o \rangle \; \mathcal{R}_{Scope} \; r' \neq undefined \qquad r_o = \mu(r')(o)}{r_s, pc \vdash \langle \mu, \mathsf{upgStruct}(o, \sigma), \Gamma \rangle \Downarrow_{IF} \langle \mu, undefined, \Gamma\left[r_o \mapsto \Gamma(r_o) \sqcup \sigma\right], pc \rangle}$$

Fig. 2. JavaScript Monitored Semantics

FUNCTION LITERAL

$$\frac{o \doteq [@proto \mapsto \#objProt] \qquad r_f, r_p \notin \mathsf{dom}(\mu) \qquad o_f \doteq [@fscope \mapsto r, @code \mapsto \lambda x.s, prototype \mapsto r_p, @proto \mapsto null] \qquad \mu' \doteq \mu\,[r_p \mapsto o, r_f \mapsto o_f] \qquad \Gamma' \doteq \Gamma \begin{bmatrix} (r_f, @fscope) \mapsto pc, (r_f, @code) \mapsto pc, (r_f, prototype) \mapsto pc, \\ (r_f, @proto) \mapsto pc, (r_p, @proto) \mapsto pc, r_f \mapsto pc, r_p \mapsto pc \end{bmatrix}}{r, pc \vdash \langle \mu, \mathsf{function}(x)\{s\}, \Gamma \rangle \Downarrow_{IF} \langle \mu', r_f, \Gamma', pc \rangle}$$

FUNCTION CALL

$$\frac{\begin{array}{c} r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_0, e_1, \Gamma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Gamma_1, \sigma_1 \rangle \\ pc' = \sigma_0 \sqcup \Gamma_1(r_0, @fscope) \qquad r' = \mu_1(r_0, @fscope) \qquad \lambda x.s \doteq \mu_1(r_0, @code) \\ r'' \notin \mathsf{dom}(\mu_1) \qquad \Gamma' \doteq \Gamma_1\left[(r'', x) \mapsto pc' \sqcup \sigma_1, (r'', @scope) \mapsto pc', (r'', @this) \mapsto pc', r'' \mapsto pc'\right] \\ o \doteq [x \mapsto v_1, @scope \mapsto r', @this \mapsto \#global] \qquad \mu' \doteq \mu_1\left[r'' \mapsto o\right] \qquad r'', pc' \vdash \langle \mu', s, \Gamma' \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle \end{array}}{r, pc \vdash \langle \mu, e_0(e_1), \Gamma \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle}$$

METHOD CALL

$$\frac{\begin{array}{c} r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_0, e_1, \Gamma_0 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Gamma_1, \sigma_1 \rangle \\ r, pc \vdash \langle \mu_1, e_2, \Gamma_1 \rangle \Downarrow_{IF} \langle \mu_2, v_2, \Gamma_2, \sigma_2 \rangle \qquad \langle \mu_2, r_0, m_1, \Gamma_2 \rangle \mathcal{R}_{Proto}\ \langle r_m, \sigma_m \rangle \\ pc' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_m \sqcup \Gamma_2(r_m, @fscope) \qquad r' \doteq \mu_2(r_m, m_1)(@fscope) \qquad \lambda x.s \doteq \mu_2(r_m, m_1)(@code) \\ o \doteq [x \mapsto v_2, @scope \mapsto r', @this \mapsto r_0] \qquad r'' \notin \mathsf{dom}(\mu_2) \qquad \mu' \doteq \mu_2\left[r'' \mapsto o\right] \\ \Gamma' \doteq \Gamma_2\left[(r'', x) \mapsto pc' \sqcup \sigma_2, (r'', @scope) \mapsto pc', (r'', @this) \mapsto pc', r'' \mapsto pc'\right] \qquad r'', pc' \vdash \langle \mu', s, \Gamma' \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle \end{array}}{r, pc \vdash \langle \mu, e_0[e_1](e_2), \Gamma \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle}$$

CONSTRUCTOR CALL

$$\frac{\begin{array}{c} r, pc \vdash \langle \mu, e_0, \Gamma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Gamma_0, \sigma_0 \rangle \qquad r, pc \vdash \langle \mu_0, e_1, \Gamma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Gamma_1, \sigma_1 \rangle \\ pc' = \sigma_0 \sqcup \Gamma_1(r_0, @fscope) \qquad r' = \mu_1(r_0, @fscope) \qquad \lambda x.s \doteq \mu_1(r_0, @code) \qquad r_p \doteq \mu_1(r_0, prototype) \\ \sigma_p \doteq \Gamma_1(r_0, prototype) \qquad o \doteq [@proto \mapsto r_p] \qquad r_o \notin \mathsf{dom}(\mu_1) \qquad r'' \notin \mathsf{dom}(\mu') \\ o_s \doteq [x \mapsto v_1, @scope \mapsto r', @this \mapsto r_o] \qquad \mu' \doteq \mu_1\left[r_o \mapsto o, r'' \mapsto o_s\right] \\ \Gamma' \doteq \Gamma_1\left[(r_o, @proto) \mapsto pc' \sqcup \sigma_p, r_o \mapsto pc', (r'', x) \mapsto pc' \sqcup \sigma_1, (r'', @scope) \mapsto pc', (r'', @this) \mapsto pc', r'' \mapsto pc'\right] \\ r'', pc' \vdash \langle \mu', s, \Gamma' \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle \end{array}}{r, pc \vdash \langle \mu, \mathsf{new}\ e_0(e_1), \Gamma \rangle \Downarrow_{IF} \langle \mu'', v, \Gamma'', \sigma \rangle}$$

Fig. 3. JavaScript Monitored Semantics

- [ASSIGNMENT-1] $pc \leq \Gamma_0(r_x, x)$. The constraint prevents sensitive upgrades by requiring that the level of variable x be higher than the level of the context (pc).
- [ASSIGNMENT-2] $pc \leq \Gamma_1(\#global)$. This rule is applied if variable x is not previously defined and thus is added to the global object. The constraint prevents sensitive upgrades by requiring that the structure security level of the global object to be higher than the level of the context (pc).
- [VARIABLE UPGRADE] $pc \leq \Gamma(r_x, x)$ which prevents a sensitive upgrade.
- [PROPERTY UPGRADE] $pc \sqcup \Gamma(r', o) \leq \Gamma(r_o, p)$. The constraint prevents the upgrade of the level of a property p via an object referred by a higher property o. (Notice however that the same property p can be upgraded from an alias variable that is also assigned to the reference that points to o, but that is labeled with a lower security level).

– [Structure Upgrade] $pc \sqcup \Gamma(r', o) \leq \Gamma(r_o)$. The constraint prevents a structure upgrade via a an object referred by a higher property o.

Non-Interferent Monitor. We say that a security monitor is non-interferent if it preserves the low-equality relation. Informally, an information flow monitor is non-interferent if whenever an attacker cannot distinguish two memories before executing a program, then the attacker cannot distinguish the final memories.

Theorem 1 (Non-Interferent Monitor). *For any program s, two memories μ and μ', respectively labeled by Γ and Γ', a reference r, and security levels pc and σ, if there exists a function β on $\mathcal{R}ef$ such that $\mu, \Gamma \approx_{\beta,\sigma} \mu', \Gamma'$ and:*

$$r, pc \vdash \langle \mu, s, \Gamma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Gamma_f, \sigma_f \rangle \quad \beta(r), pc \vdash \langle \mu', s, \Gamma' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Gamma'_f, \sigma'_f \rangle$$

Then, there exists a function β' such that $\beta \leq \beta'$, $\mu_f, \Gamma_f \approx_{\beta',\sigma} \mu'_f, \Gamma'_f$ and $v_f, \sigma_f \approx_{\beta',\sigma} v'_f, \sigma'_f$.

Example 9 (Blocked Execution). Observe that the execution of the program given in Example 6 is not allowed to go through when $h = 1$, because one cannot add a property to an object in a high context if the object (as it is the case) has a low structure security level. That would constitute a sensitive upgrade. For that example to be legal, one has to upgrade the structure security level of the object before executing the if statement.

3 Monitor Inlining

This section presents a new inlining compiler for instrumenting JavaScript programs in order to simulate their execution in the monitored semantics introduced in Section 2. This instrumentation rests on a technique that consists in pairing up each variable/property with a new one, called *shadow variable/shadow property* [8, 16], that holds its corresponding security level. Additionally, the level of the current context is always available in a special local variable named $\$pc$. Since the compiled program has to handle security levels, we choose to include the set of security levels 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 the order relation and to the least upper bound between security levels ($\leq$ and $\sqcup$).

We assume given a set of variable and property names that does not intercept with those available for the programmer. In particular, in the compilation of some kinds of expressions, the compiler uses two extra variables that are meant to store the value and the security level of the original expression, so that they can be later used in the compilation of other expressions that include it. Hence, we assume that the set of compiler variables includes two infinite sets of indexed variables:

– $\{\$\hat{l}_i\}_{i \in \mathbf{N}}$ used to store the levels of intermediate expressions during execution,

– $\{\$\hat{v}_i\}_{i \in \mathbf{N}}$ used to store the values of intermediate expressions during execution.

For each variable/property the compiler adds a new *shadow* variable/property to hold its security level during execution. Given a variable x, we denote the corresponding shadow variable by $\$l_x$. In contrast to variables, whose names are available during compilation, property names are dynamically computed. Hence, we assume the existence of a library function $\$shadow$ that is available during execution and that, given a property name, dynamically computes the name of the corresponding shadow property. For simplicity, all identifiers reserved for the compiler are prefixed with a dollar sign, \$. At the implementation level, we ensure that compiler identifiers do not overlap with those of the original program by generating them randomly.

We consider two special types of expressions:

– *Simple Expressions:* comprising identifiers, values, the *this* keyword and binary operations. The evaluation of simple expressions does not change the memory and therefore simple expressions are preserved by compilation. We use $\hat{e}$ to denote a simple expression.
– *Indexed Expressions:* comprising property look-up expressions, function calls, method calls, constructor calls, and function literals. Each indexed expression is compiled to a statement that assigns its level to a variable $\$\hat{l}_i$ and its value to a variable $\$\hat{v}_i$, where i is the index of the expression.

In the specification of the compiler, we assume that in certain contexts (for instances, the arguments of a function) there can only appear simple expressions. This means that the application of our inlining compiler must be preceded by a normalization step.

Besides adding to every object o an additional property $\$l_p$ for each property p in the original domain of o (that is used to store the security level associated with p), the inlined monitoring code also adds to o a special property $\$struct$ that is used to store its current structure security level.

Example 10 (Instrumented Memory versus Labeled Memory). Suppose that an object $o = [p \mapsto v_0, q \mapsto v_1]$ pointed by r_o is labeled by Γ in such a way that $\Gamma|_{r_o} = [p \mapsto H, q \mapsto L]$ and $\Gamma(r_o) = L$. The instrumented counterpart of object o labeled by Γ is given by $\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$struct \mapsto L]$.

Extending JavaScript Semantics. In the specification of the compiler, we assume the existence of a binary operator hasOwnProp that tests if the object given as its left operand defines the property given as its right operand. In practice, this operator does not exist; instead, there is a method *hasOwnProperty*, which is accessible to every object *via* its corresponding prototype chain, that tests if the object on which it is invoked defines the property given as its argument. We chose not to model this feature of the language exactly as it is in practice in order to keep the model as simple as possible. Doing it otherwise would imply cluttering the already complex semantics of the language by having an

alternative case for the Rule [Method Call], which would specify the semantics of the *hasOwnProperty* method call. Still, in order to simplify the specification of the compiler we extend the syntax and semantics of the language with the object literal notation {}, that creates a new object with a *null* prototype. The extension of the semantic relation $\Downarrow_{JS}$ for both hasOwnProp and {} is given in Figure 4. Modeling the *hasOwnProperty* method as an operator rather than a

$$\textsf{hasOwnProp - TRUE}\quad \frac{r \vdash \langle \mu, e_0 \rangle \Downarrow_{JS} \langle \mu, r \rangle \qquad r \vdash \langle \mu, e_1 \rangle \Downarrow_{JS} \langle \mu, m \rangle \qquad m \in \mathsf{dom}(\mu(r))}{r \vdash \langle \mu, e_0 \ \mathsf{hasOwnProp}\ e_1) \rangle \Downarrow_{JS} \langle \mu, true \rangle}$$

$$\textsf{hasOwnProp - FALSE}\quad \frac{r \vdash \langle \mu, e_0 \rangle \Downarrow_{JS} \langle \mu, r \rangle \qquad r \vdash \langle \mu, e_1 \rangle \Downarrow_{JS} \langle \mu, m \rangle \qquad m \notin \mathsf{dom}(\mu(r))}{r \vdash \langle \mu, (e_0 \ \mathsf{hasOwnProp}\ e_1) \rangle \Downarrow_{JS} \langle \mu, false \rangle}$$

$$\textsc{Object Literal}\quad \frac{o \doteq [@proto \mapsto null] \qquad r_o \notin \mathsf{dom}(\mu)}{r \vdash \langle \mu, \{\} \rangle \Downarrow_{JS} \langle \mu \left[r_o \mapsto o \right], r_o \rangle}$$

Fig. 4. Extension of $\Downarrow_{JS}$ for hasOwnProp and {}

method dismisses the possibility of having malicious code interfering with it. In Section 4, we explain how we deal with this problem in the implementation.

Formal Specification. In order to simplify the definition of the main compiler, we introduce three auxiliary compilation functions that are used by the main one.

- $\mathcal{C}_l\langle . \rangle$ compiles each simple expression to an expression that evaluates to its level.
- $\mathcal{C}_{enf}\langle ., . \rangle$ receives as input two expressions that evaluate to two security levels, σ_0 and σ_1, and outputs a statement that terminates if and only if $\sigma_0 \leq \sigma_1$.
- $\mathcal{C}_{lkup}\langle ., ., . \rangle$ receives as input a simple expression that evaluates to a reference r, a simple expression that evaluates to a string m, and an index i, and outputs a statement that simulates the instrumented Prototype-Chain Look-Up semantic relation (Definition 2) by storing the reference of the object that is closest to r in the corresponding prototype-chain and that has a binding for m in variable $\$\hat{v}_i$ and the level associated with the look-up process in variable $\$\hat{l}_i$.

The formal definitions of $\mathcal{C}_l\langle . \rangle$, $\mathcal{C}_{enf}\langle ., . \rangle$, and $\mathcal{C}_{lkup}\langle ., ., . \rangle$ are given in Figure 5.

Simulating the Prototype-Chain Look-Up semantic relation requires being able to inspect the prototype chain of any given object in order to find the

object that is closest to it and that defines a binding for the searched property. However, JavaScript does not allow the programmer to access the prototype of an object directly (because the @*proto* property is internal and therefore not available programmatically). In order to cope with this issue, instrumented constructor calls have an additional argument corresponding to the prototype of the object being created. In the body of the constructor, the prototype of the object being created is assigned to its property \$*proto*. Hence, for every reference r pointing to an instrumented object $\mu(r)(\$proto) = \mu(r)(@proto)$.

VARIABLE
$\mathcal{C}_l\langle x\rangle = \$pc \sqcup \$l_x$

VALUE
$\mathcal{C}_l\langle v\rangle = \$pc$

THIS
$\mathcal{C}_l\langle this\rangle = \$pc$

BINARY OPERATION
$\mathcal{C}_l\langle \hat{e}_1 \ \mathsf{op}_2 \ \hat{e}_2\rangle = \mathcal{C}_l\langle \hat{e}_1\rangle \sqcup \mathcal{C}_l\langle \hat{e}_2\rangle$

ENFORCEMENT
$\mathcal{C}_{enf}\langle \hat{e}_0, \hat{e}_1\rangle = \mathsf{if}(\hat{e}_0 \leq \hat{e}_1)\{null;\}\ \mathsf{else}\ \{\$diverge();\}$

LOOK-UP

$$\mathcal{C}_{lkup}\langle \hat{e}_0, \hat{e}_1, i\rangle = \begin{cases} \$\hat{l}_i = \bot; \\ \$\hat{v}_i = \hat{e}_0; \\ \mathsf{while}(!(\$\hat{v}_i\ \mathsf{hasOwnProp}\ \hat{e}_1))\{ \\ \quad \$\hat{l}_i = \$\hat{l}_i \sqcup \$\hat{v}_i.\$l_{@proto} \sqcup \$\hat{v}_i.\$struct; \\ \quad \$\hat{v}_i = \$\hat{v}_i.\$proto; \\ \} \\ \mathsf{if}(\$\hat{v}_i)\{\$\hat{l}_i = \$\hat{l}_i \sqcup \$\hat{v}_i[\$shadow(\hat{e}_1)];\}\ \mathsf{else}\ \{null;\} \end{cases}$$

Fig. 5. Auxiliary Compilers

The main compiler is defined as a function $\mathcal{C}\langle . \rangle$ on expressions and statements and is presented in Figure 6. We omit the rules for the compilation of the while statement, the if statement, the sequence statement, and the expression statement, because they coincide with those of previous work on inlining information flow security monitors [16]. Besides the three auxiliary compilation functions already introduced, the compiler uses an additional compilation function, $\mathcal{C}_{fun}\langle . \rangle$, that maps each function literal to its compiled counterpart. Since these two compilation functions are interdependent we present them together. The instrumented code generated by the compiler makes use of two runtime functions \$*shadow* and \$*diverge*. Function \$*shadow* receives as input an arbitrary string corresponding to a property name and outputs the name of the corresponding shadow property. Function \$*diverge* never terminates. In order to prove the correctness of the compiler one has to assume that these runtime functions behave as expected. Assumptions 1 and 2 formally state these claims.

Assumption 1 (Semantics of \$*shadow*) *Given a simple expression $\hat{e}$, a memory μ, a reference r, and a property p such that $r \vdash \langle \mu, \hat{e}\rangle \Downarrow_{JS} \langle \mu, p\rangle$, then the following holds: $r \vdash \langle \mu, \$shadow(\hat{e})\rangle \Downarrow_{JS} \langle \mu, \$l_p\rangle$.*

Assumption 2 (Semantics of $\$diverge$) *For any memory μ and reference r, $r \vdash \langle \mu, \$diverge() \rangle \Uparrow_{JS}$.*

Function $\mathcal{C}_{fun}\langle . \rangle$ instruments function literals. After executing a function, or a method, the instrumented code that calls that function/method must not only receive the return value of the original function/method but also the level that is to be associated with that value. Hence, compiled function literals return 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. Additionally, when executing a function, a method, or a constructor, the calling code must be able to specify which is the level to be associated with the argument and which is the level of the current context. Hence, every compiled function receives two extra parameters corresponding respectively to the level of the first parameter and to the level of the context in which it is called. Finally, the evaluation of every compiled function starts by checking it the function which is executing is being called as a constructor. To check this, it verifies if its fourth argument ($\$proto$) is defined, in which case it initializes the shadow properties of the newly created object (namely, $@proto$ and $@struct$). Moreover, it assigns its property $\$proto$ to the corresponding prototype (in order to be able to simulate the Prototype Chain Look-Up Semantic Relation as discussed above).

Correctness. The theorem of correctness states that a program terminates when starting from a given configuration in the monitored semantics if and only if the compiled counterpart also terminates in a *similar* configuration. In order to state the correctness theorem, we need to formally define a similarity relation between labeled memories in the monitored semantics and instrumented memories in the JavaScript semantics. The proposed similarity relation requires that, for every object in the labeled memory, the corresponding labeling coincide with the instrumented labeling (except for the levels of some internal properties that are not needed because they can be inferred from other levels) 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, which differs from the one defined in Section 2 in that it relates each function literal with its compilation by $\mathcal{C}_{fun}$ and allows the domain of the instrumented object to be larger than that of the original one. The similarity relation is given is Definition 4.

Definition 4 (Memory Similarity). *A memory μ labeled by Γ is similar to a memory μ' w.r.t. β, written $\mu, \Gamma \ \mathcal{R}_\beta \ \mu'$, iff $\mathsf{dom}(\beta) = \mathsf{dom}(\mu)$ and for all $r \in \mathsf{dom}(\beta)$ where $o = \mu(r)$ and $o' = \mu'(\beta(r))$, $\Gamma(r) = o'(\$struct)$ and:*

$$\forall p \in \mathsf{dom}(o) \backslash \{@scope, @this, @code\} \cdot o(p) \sim_{\beta, \mathcal{C}_{fun}} o'(p) \ \wedge \ \Gamma(r, p) = o'(\$l_p)$$

Theorem 2 (Correctness). *Given a reference mapping β, a labeled configuration $\langle \mu, s, \Gamma \rangle$, a configuration $\langle \mu', \mathcal{C}_{\mathcal{P}}\langle s \rangle \rangle$, two scope references r and $\beta(r)$ in μ and μ' resp., and a security level $pc = \mu'(\beta(r))(\$pc)$, such that $\mu, \Gamma \ \mathcal{R}_\beta \ \mu'$; then there exists $\langle \mu_f, v_f, \Gamma_f, \sigma \rangle$ such that $r, pc \vdash \langle \mu, s, \Gamma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Gamma_f, \sigma \rangle$ if*

FUNCTION LITERAL - EXPRESSION

$$\frac{s_0 = \begin{cases} \mathsf{if}(\$proto)\{ \\ \quad this.\$proto = \$proto; \\ \quad this.\$l_{@proto} = \$pc; \\ \quad this.\$struct = \$pc; \\ \} \ \mathsf{else}\ \{null;\} \end{cases} \qquad s_1 = \begin{cases} \mathcal{C}\langle s\rangle; \\ \$ret = \{\}; \\ \$ret.\$v = \hat{e}; \\ \$ret.\$l = \mathcal{C}_l\langle\hat{e}\rangle; \\ \$ret \end{cases}}{\mathcal{C}_{fun}\langle \mathsf{function}(x)\{s;\hat{e}\}^i\rangle = \mathsf{function}(x, \$l_x, \$pc, \$proto)\{s_0; s_1\}}$$

PROPERTY UPDATE/CREATION

$$\mathcal{C}\langle\hat{e}_0[\hat{e}_1] = \hat{e}_2\rangle = \begin{cases} \mathsf{if}(\hat{e}_0\ \mathsf{hasOwnProp}\ \hat{e}_1)\ \{ \\ \quad \mathcal{C}_{enf}\langle\mathcal{C}_l\langle\hat{e}_0\rangle \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle, \hat{e}_0[\$shadow(\hat{e}_1)]\rangle \\ \}\ \mathsf{else}\ \{ \\ \quad \mathcal{C}_{enf}\langle\mathcal{C}_l\langle\hat{e}_0\rangle \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle, \hat{e}_0.\$struct\rangle \\ \} \\ \hat{e}_0[\$shadow(\hat{e}_1)] = \mathcal{C}_l\langle\hat{e}_0\rangle \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle \sqcup \mathcal{C}_l\langle\hat{e}_2\rangle; \\ \hat{e}_0[\hat{e}_1] = \hat{e}_2; \end{cases}$$

FUNCTION LITERAL - STATEMENT

$$\mathcal{C}\langle\mathsf{function}(x)\{s;\hat{e}\}^i\rangle = \begin{cases} \$\hat{v}_i = \mathcal{C}_{fun}\langle\mathsf{function}(x)\{s;\hat{e}\}\rangle; \\ \$\hat{v}_i.\$l_{prototype} = \$pc; \\ \$\hat{v}_i.\$struct = \$pc; \\ \$\hat{v}_i.\$l_{@fscope} = \$pc; \\ \$\hat{v}_i.prototype.\$proto = null; \\ \$\hat{v}_i.prototype.\$l_{@proto} = \$pc; \\ \$\hat{v}_i.prototype.\$struct = \$pc; \\ \$\hat{l}_i = \$pc; \end{cases}$$

METHOD CALL

$$\mathcal{C}\langle\hat{e}_0[\hat{e}_1](\hat{e}_2)^i\rangle = \begin{cases} \mathcal{C}_{lkup}\langle\hat{e}_0, \hat{e}_1, i\rangle \\ \$\hat{l}_i = \$\hat{l}_i \sqcup \$\hat{v}_i.\$l_{@fscope}; \\ \$\hat{l}_{ctx} = \mathcal{C}_l\langle\hat{e}_0\rangle \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle \sqcup \$\hat{l}_i; \\ \$\hat{l}_{arg} = \$\hat{l}_{ctx} \sqcup \mathcal{C}_l\langle\hat{e}_2\rangle; \\ \$ret = \hat{e}_0[\hat{e}_1](\hat{e}_2, \$\hat{l}_{arg}, \$\hat{l}_{ctx}); \\ \$\hat{v}_i = \$ret[\text{“\$v”}]; \\ \$\hat{l}_i = \$ret[\text{“\$l”}]; \end{cases}$$

FUNCTION CALL

$$\mathcal{C}\langle\hat{e}_0(\hat{e}_1)^i\rangle = \begin{cases} \$\hat{l}_{ctx} = \hat{e}_0.\$l_{@fscope} \sqcup \mathcal{C}_l\langle\hat{e}_0\rangle; \\ \$\hat{l}_{arg} = \$\hat{l}_{ctx} \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle; \\ \$ret = \hat{e}_0(\hat{e}_1, \$\hat{l}_{arg}, \$\hat{l}_{ctx}); \\ \$\hat{v}_i = \$ret[\text{“\$v”}]; \\ \$\hat{l}_i = \$ret[\text{“\$l”}]; \end{cases}$$

CONSTRUCTOR CALL

$$\mathcal{C}\langle\mathsf{new}\ \hat{e}_0(\hat{e}_1)^i\rangle = \begin{cases} \$\hat{l}_{ctx} = \hat{e}_0.\$l_{@fscope} \sqcup \mathcal{C}_l\langle\hat{e}_0\rangle; \\ \$\hat{l}_{arg} = \$\hat{l}_{ctx} \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle; \\ \$proto = \hat{e}_0.prototype; \\ \$\hat{v}_i = \mathsf{new}\ \hat{e}_0(\hat{e}_1, \$\hat{l}_{arg}, \$\hat{l}_{ctx}, \$proto); \\ \$\hat{l}_i = \$\hat{l}_{ctx}; \end{cases}$$

PROPERTY LOOK-UP

$$\mathcal{C}\langle\hat{e}_0[\hat{e}_1]^i\rangle = \begin{cases} \mathcal{C}_{lkup}\langle\hat{e}_0, \hat{e}_1, i\rangle \\ \$\hat{v}_i = \$\hat{v}_i[\hat{e}_1]; \\ \$\hat{l}_i = \mathcal{C}_l\langle\hat{e}_0\rangle \sqcup \mathcal{C}_l\langle\hat{e}_1\rangle \sqcup \$\hat{l}_i; \end{cases}$$

SIMPLE EXPRESSION

$$\mathcal{C}\langle\hat{e}\rangle = \hat{e};$$

SIMPLE EXPRESSION ASSIGNMENT

$$\mathcal{C}\langle x = \hat{e}\rangle = \begin{cases} \mathcal{C}_{enf}\langle\$pc, \$l_x\rangle; \\ \$l_x = \mathcal{C}_l\langle x\rangle; \\ x = \hat{e}; \end{cases}$$

INDEXED EXPRESSION ASSIGNMENT

$$\frac{i = \mathsf{index}(\bar{e})}{\mathcal{C}\langle x = \bar{e}\rangle = \begin{cases} \mathcal{C}\langle\bar{e}\rangle \\ \mathcal{C}_{enf}\langle\$pc, \$l_x\rangle; \\ \$l_x = \$\hat{l}_i; \\ x = \$\hat{v}_i; \end{cases}}$$

UPGRADE PROPERTY

$$\mathcal{C}\langle\mathsf{upgProp}(o, p, \sigma)\rangle = \begin{cases} \mathsf{if}(o\ \mathsf{hasOwnProp}\ p)\ \{ \\ \quad \mathcal{C}_{enf}\langle\mathcal{C}_l\langle o\rangle, o.\$l_p\rangle; \\ \quad o.\$l_p = o.\$l_p \sqcup \sigma; \\ \}\ \mathsf{else}\ \{\$diverge();\} \end{cases}$$

UPGRADE STRUCTURE

$$\mathcal{C}\langle\mathsf{upgStruct}(o, \sigma)\rangle = \begin{cases} \mathcal{C}_{enf}\langle\mathcal{C}_l\langle o\rangle, o.\$struct\rangle \\ o.\$struct = o.\$struct \sqcup \sigma; \end{cases}$$

UPGRADE VARIABLE

$$\mathcal{C}\langle\mathsf{upgVar}(x, \sigma)\rangle = \begin{cases} \mathcal{C}_{enf}\langle\$pc, \$l_x\rangle \\ \$l_x = \$l_x \sqcup \sigma; \end{cases}$$

Fig. 6. Information Flow Monitor Inlining Compiler

and only if there exists $\langle \mu'_f, v'_f \rangle$ *such that* $\beta(r) \vdash \langle \mu', \mathcal{C}_{\mathcal{P}}\langle s \rangle \rangle \Downarrow_{JS} \langle \mu'_f, v'_f \rangle$. *If both configurations converge, then there is a function* β' *extending* β *such that* $\mu_f, \Gamma_f \; \mathcal{R}_{\beta'} \; \mu'_f$, $pc = \mu'_f(\beta(r))(\$pc)$, *and if* s *is an indexed expression with index* i: $v_f \sim_{\beta', \mathcal{C}_{fun}} \mu'_f(\beta(r))(\$\hat{v}_i)$ *and* $\mu'_f(\beta(r))(\$\hat{l}_i) = \sigma$.

4 Examples for Dealing with Untrusted Code

The compiler prototype is implemented in JavaScript and is available online at [1] together with a broad set of examples that includes those of the paper and with the full version of this paper (which includes the main proofs). Here, we discuss implementation details regarding how the compiler deals with particular features of the JavaScript semantics and with malicious code.

Malicious code. The correctness of the instrumentation relies on the assumption that the internal variables and object properties used by the compiler do not intercept with those of the program to be compiled. Naturally, a malicious program may try to bypass the runtime enforcement mechanisms generated by the compiler by rewriting some of its internal runtime variables. For example, suppose that the lattice to be used is bound to a variable $\$lat$. A malicious program can exploit this information to "shut down" all the runtime verifications added by the compiler. To do this, it suffices to assign to $\$lat$ the most permissive security lattice: $\$lat = most\_permissive\_lattice$. After doing this, all the constraints added by the compiler are trivially verified. In order to prevent this kind of malicious behaviour, variable and property names used by the compiler are randomly generated. In order to tamper with the compiler runtimes, the attacker code must be able to guess the randomly generated names.

The names of internal local variables (like $\$pc$) are randomly generated, whereas those of global variables (like $\$lat$) are assumed to be properties of a single object that is accessed through a global variable whose name is also randomly generated. In this way, the runtime libraries that are assumed to be available during the execution of compiled code do not need to be dynamically computed.

Type coercions . Malicious code can exploit implicit type coercions to compromise the security of compiled code, as one can see in the example below.

Example 11 (Malicious Code Example - Exploiting Implicit Type Coercions).

```
1 o1.toString = function() { return 'p'; };
2 o2.p = secret;
3 public = o2[o1];
```

Since $o1$ appears in a context in which the JavaScript interpreter is expecting a string, its *toString* method is invoked.

Our instrumentation disallows any kind of implicit type coercion. Since relying in implicit type coercions is considered a bad programming practice [9] that is error-prone and hinders maintainability, we do not find this restriction a serious shortcoming of the compiler.

Native functions. The compiler correctness does not rely on any kind of function that is liable to malicious code, namely native functions.

Example 12 (Malicious Code Example - Tampering with native functions).

```
1 o.p = 0;
2 upgStruct(o, H);
3 o.hasOwnProperty = function () { return false}
4 if(h) { o.p = 1;}
```

The example above is illegal, because updating the value of a low property in a high context constitutes a sensitive upgrade. Creating a new property in a high context is, however, allowed in this example, because o has a high structure security level. Naturally, the monitoring code inlined by the compiler must test if the object defines the property that is being set in order to decide which constraint to apply. This program tries to bypass the inlined monitoring code by redefining the *hasOwnProperty* method of object o. Therefore, if the inlined monitoring code generated by the compiler uses o's own *hasOwnProperty* method, the execution of this program is allowed to go through, which entails a security violation.

The monitoring code generated by our compiler does not rely on any native function that is liable to malicious code. Instead, when setting up the compiler runtimes, the runtime enforcement mechanism "saves" all original native functions that it will potentially use later as properties of an object stored in a global variable whose name is randomly generated.

Example 13 (Malicious Code Example - Protection against the tampering of native functions).

```
1 _runtime.hasOwnProperty = (function(o){
2    return function(o, p) {
3       return o.hasOwnProperty.call(o, p);
4    }
5 })({});
```

Handling functions whose code is not available for instrumentation. In our implementation, we provide a way to deal with functions, like native functions, that are provided to the program by the runtime environment, but whose code is not available and thus cannot be instrumented. Our approach to managing this kind of functions consists in specifying for each such function f the following JavaScript functions:

- *isInDomain* returns true when it receives f as input;
- *enforce* checks if the constraints associated with the invocation of f are verified;
- *updtLabeling* updates the instrumented labeling and outputs the reading effect associated with the call to f;
- *processArg* pre-processes the argument given to f;

– *processRet* post-processes the return value of f.

We call this set of functions the *IFlow Signature* of f. Function calls are therefore compiled in the following way:

$$\mathcal{C}\langle \hat{e}_0(\hat{e}_1)^i]\rangle = \begin{cases} \$l_{ctxt} = \mathcal{C}_l\langle \hat{e}_0\rangle;\ \$if_{sig} = \$nativeRegister(\hat{e}_0); \\ \mathsf{if}(!\$if_{sig})\ \{\mathcal{C}\langle \hat{e}_0(\hat{e}_1)^i\rangle\}\ \mathsf{else}\ \{ \\ \quad \$if_{sig}.enforce(\hat{e}_0, \hat{e}_1, \$l_{ctxt}, \mathcal{C}_l\langle \hat{e}_1\rangle); \\ \quad \$\hat{v}_i = \$if_{sig}.processRet(\hat{e}_0(\$if_{sig}.processArg(\hat{e}_1))); \\ \quad \$\hat{l}_i = \$if_{sig}.updtLabeling(\$\hat{v}_i, \hat{e}_0, \hat{e}_1, \$l_{ctxt}, \mathcal{C}_l\langle \hat{e}_1\rangle); \\ \} \end{cases}$$

Below, we give examples of possible IFlow Signatures for *eval* and *setTimeOut*.

Example 14 (IFlow Signature for eval). For the case of ***eval***, we propose an IFlow signature in which the *processArg* function is used to compile the argument that is passed to eval at runtime before proceeding to its evaluation. This is equivalent to on-the-fly inlining as proposed by Magazinius [15].

```
1 function processArg(str) { return $compile(str);  }
```

Observe that in order for this IFlow Signature to work, the compiler itself must be part of the runtime libraries required for running instrumented code.

Example 15 (IFlow Signature for setTimeOut). The program below schedules the execution of a statement that updates the value of a low confidentiality variable, depending on the value of a high variable. This constitutes a sensitive upgrade. The IFlow Signature for *setTimeout* must do more than compile the code that is to be executed, it must wrap it in a function literal and set the default level of its program counter to the current $\$pc$ level. By doing this, when this code is finally executed, the $\$pc$ level is be high and the assignment that constitutes a sensitive upgrade is prevented.

```
1 var x = 0;
2 if (h) { setTimeout('x = 4', 2000);}
```

5 Conclusion

We have presented a sound information flow monitor for JavaScript, as well as a program transformation that inlines our information flow monitor. Having implemented a prototype of the compiler [1], we show its effectiveness in a number of examples including examples that can leak information due to non-standard JavaScript semantic features as implicit type coercions.

The correctness of the instrumentation relies on the assumption that identifier names used by the compiler do not overlap with those of the code to be compiled. In the implementation, we remove this assumption by randomizing the identifier names associated with the compiler's runtime libraries. It would be interesting to use the techniques of [3] for the randomizing compiler to prove a stronger result w.r.t. untrusted code [11]. Proofs can be found in [1].

References

1. Information flow instrumentation 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. Martín Abadi and Jérémy Planul. On layout randomization for arrays and functions. In David A. Basin and John C. Mitchell, editors, *POST*, volume 7796 of *Lecture Notes in Computer Science*, pages 167–185. Springer, 2013.
4. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In Stephen Chong and David A. Naumann, editors, *PLAS*, pages 113–124. ACM, 2009.
5. Thomas H. Austin and Cormac Flanagan. Multiple Facets for Dynamic Information Flow. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.
6. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW*, pages 253–. IEEE Computer Society, 2002.
7. Cenzic Inc. Web application security trends report Q1, 2013. `http://www.cenzic.com/`, 2013.
8. Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *CSF*. IEEE Computer Society, 2010.
9. Douglas Crockford. *JavaScript: The Good Parts*. O Reilly, 2008.
10. Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, 2010.
11. Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In George C. Necula and Philip Wadler, editors, *POPL*, pages 323–335. ACM, 2008.
12. Daniel Hedin and Andrei Sabelfeld. Information-Flow Security for a Core of JavaScript. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2012.
13. Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 270–283, 2010.
14. Gurvan Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
15. Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. In Kai Rannenberg, Vijay Varadharajan, and Christian Weber, editors, *SEC*, volume 330 of *IFIP Advances in Information and Communication Technology*, pages 173–186. Springer, 2010.
16. Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
17. V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer, 2006.