Information-Flow Security for a Core of JavaScript

Daniel Hedin Andrei Sabelfeld
Chalmers University of Technology, Gothenburg, Sweden

***Abstract*—Tracking information flow in dynamic languages remains an important and intricate problem. This paper makes substantial headway toward understanding the main challenges and resolving them. We identify language constructs that constitute a core of JavaScript: objects, higher-order functions, exceptions, and dynamic code evaluation. The core is powerful enough to naturally encode native constructs as arrays, as well as functionalities of JavaScript's API from the document object model (DOM) related to document tree manipulation and event processing. As the main contribution, we develop a dynamic type system that guarantees information-flow security for this language.**

I. Introduction

Tracking information flow in dynamic languages remains an important and intricate problem. The goal of this paper is understanding the fundamentals of challenges for tracking information flow in dynamic languages and making substantial headway to resolving them. We believe that with these challenges addressed, our work opens up opportunities for practical security of dynamic applications that are hard to secure with existing approaches.

Motivation: Modern web applications make increasingly intensive use of service composition. A particularly thriving service composition technique is client-side script inclusion. It allows straightforward integration of services, often provided by third parties, in so called *web mashups*. Popular examples of integrated services are Google Maps for visualizing data on interactive maps and Google Analytics for collecting statistics of web site usage. Extreme cases of third-party script inclusion can be found on sensitive web sites like barclays.es, expedia.com, scrive.se, komplett.se, and webhallen.se, where the scripts are embedded at top level, often on the same page where the user enters login credentials. With user credentials at hand, integrated client scripts have unrestricted power to engage in interaction with the hosting service. A directory for mashups at programmableweb.com contains more than 6000 registered mashups and more than 5000 registered content provider API's.

The state of the art in web mashup security calls for improvement. The *same-origin policy (SOP)*, enforced by most browsers today, is intended to restrict access between scripts from different Internet domains. SOP offers an all-or-nothing choice when including a script: either isolation, when the script is loaded in an iframe, or full integration, when the script is included in the document via a script tag.

Securing mashups is subject to currently ongoing research efforts. A recent survey of the area [38] identifies a number of approaches ranging from isolation of components to their full integration. The focus of this paper is *tight yet secure integration* for scenarios when isolation is too restrictive and full integration is insecure. In particular, we are interested in applications where access-control policies fall short and where fine-grained information-flow control is desired.

Scenarios: To illustrate such applications, consider the following scenarios. The scenarios are modeled on existing web pages, reflecting pervasive usage of advertising and statistics services on today's web.

Online advertisement: The first scenario is that of online shopping. The online shopping code includes a third-party extension for dynamically generated ads. As items are put in the shopping cart, contextually dependent ads are displayed to the user. The ads are tightly integrated in the shopping cart, mixing items with discount information, offers from the merchant, and ads from cooperating third parties. This is achieved by programmatic manipulation by JavaScript of the nodes in the *document object model* (DOM) tree, a tree representation of the underlying document. This results in a highly dynamic information flow, where secret and public information is mixed in the same linear structure.

User tracking: The second scenario is that of a login to a bank. It illustrates event handling in terms of a keypad for entering a shared secret. The keypad is part of an application that contains third-party code for creating click heat maps. Such third-party code is common on many web sites. Consequently, the position of the clicks must be kept secret from the heat map generator, since the positions of the clicks encode information about what is entered via the key pad. This implies that information flow through JavaScript event handlers must be secured.

These scenarios motivate fine-grained security that is more permissive than full isolation (as provided by, e.g., iframes) and at the same time more secure than full integration (as provided by script inclusion with no additional measures). Hence, our objective is to provide a solid foundation for information-flow control to protect the user from malicious scripts run in a browser. To this end, the paper delivers the following contributions.

Contributions: We identify a language that constitutes a core of JavaScript and includes the following constructs: objects, higher-order functions, exceptions, and dynamic code evaluation. We argue that the choice of the core captures the

essence of the language, and allows us to concentrate on the fundamental challenges for securing information flow in dynamic languages (Section II). While we address the major challenge of handling dynamic language constructs, we also resolve minor challenges associated with JavaScript. The semantics of the language closely follows the ECMA-262 standard (v.5) [18] on the language constructs from the core.

We develop a dynamic type system for information-flow security for the core language (Section III) and establish that the type system enforces a security policy of *noninterference* [11], [22], that prohibits information leaks from the program's secret sources to the program's public sinks (Section IV). Corner cases of the soundness proof are formalized in the proof assistant Coq [12]. We have implemented the type system and performed a proof-of-concept indicative study of permissiveness using the scenarios above.

We show that the core is powerful enough to naturally encode native constructs as arrays, as well as functionalities of JavaScript's DOM API related to document tree manipulation and event processing (Section V). We develop these encodings as a part of our implementation for the two scenarios above.

The first encoding implements parts of the core functionality of the Document Object Model (DOM) [27] without imposing any constraints on the navigation. In particular, direct indexing via the *childNodes* array, and node relocation when repeatedly adding the same node is fully supported. The implementation only deals with the dynamic tree structure of the DOM, in order to investigate the permissiveness of dynamic information flow. It does not obviate the need for specific handling of the DOM in the form of, e.g., wrappers implementing specific type rules, since the DOM is typically not implemented in JavaScript.

JavaScript does not include concurrency features. However, the DOM API allows creating JavaScript handlers that are triggered by external events and run without preemption. The second encoding implements a simplified version of the events of JavaScript, and shows how the propagation of events can be controlled, including how events can be stopped, the presence of certain events can be made secret, or how the presence of events can be public, while the data of the event remains secret.

II. Challenges and our approach

The ultimate goal of this work is to enforce information-flow security in JavaScript. This section outlines the main challenges and illustrates our approach to resolving them.

A. Language challenges

JavaScript is a dynamically typed, object-based language with higher-order functions, and dynamic code evaluation via, e.g., the *eval* construction. The voluminous ECMA-262 standard describes the syntax and semantics of JavaScript. The challenge is identifying a subset of the standard that captures the core of JavaScript from an information-flow perspective, i.e., where the left-out features are either expressible in the core, or where their inclusion does not reveal new information-flow challenges. We identify the core of the language to be: objects, higher-order functions, exceptions, and *eval*. In addition, we show that the well-known problems associated with the JavaScript variable handling and the *with* construct can be handled via a faithful modeling in terms of objects.

B. Security challenges

Before presenting the security challenges, we briefly recap standard information-flow terminology [15]. Traditionally, information flow analyses distinguish between *explicit* and *implicit* flows.

Explicit flows amount to directly copying information, e.g., via an explicit assignment like `l = h;`, where the value of a secret (or *high*) variable h is copied into a public (or *low*) variable l. High and low represent *security levels*. We assume two levels without losing generality: our results apply to arbitrary security lattices [14]. *Security labels* are used to store security levels for associated variables. In the following let h and l represent high and low variables.

Implicit flows may arise when the *control flow* of the program is dependent on secrets. Consider the program **if** (h) l = 1; **else** l = 0;. Depending on the secret stored in variable h, variable l will be set to either 1 or 0, reflecting the value of h. Hence, there is an implicit flow from h into l. Frequently, this is tracked by associating a security label, the *security context*, with the control flow. Above, the body of the conditional is executed in a *secret context*. Equivalently, the branches of the conditional are said to be under *secret control*. The definition of *public context* and *public control* are natural duals.

Eval: The runtime possibility to parse and execute a string in a JavaScript program, provided by the *eval* instruction, poses a critical challenge for static program analyses, since the string may not be available to a static analysis.

Our approach: Dynamic analyses do not share this limitation, by virtue of the fact that dynamic analyses are executed at runtime. This is one of the two main reasons for the use of a dynamic type system in this paper.

Aliasing and flow sensitivity: The type system of JavaScript is *flow sensitive*; the types of, e.g., variables and fields are allowed to vary during the execution. In addition, objects in JavaScript are heap allocated and represented by primitive references — their heap location. Hence, objects in JavaScript may be aliased. An alias occurs when two syntactically different program locations (e.g., two variables x and y) refer to the same object (contain the same primitive reference).

Flow sensitivity in the presence of aliases poses a significant challenge for static analyses, since changing the labels of one of the locations requires that the security labels of all aliased locations are changed as well, which requires the static analysis to keep track of the aliasing.

Our approach: Dynamic approaches do not share this limitation, since they are able to store the security label in the referred object instead of associating it with the syntactic program location. Consider the following program:

```
x = new Object(); x.f = 0; y = x; y.f = h;
```

First, a new object is allocated and the primitive reference to the object is stored into x. Thereafter, the field f of the newly allocated object is set to 0, and y is made an alias of x, and the field f is overwritten by a secret via y. This kind of programs are rejected by static analysis like Jif [32]. In the dynamic setting, we simply update the security label of the value of the field f of the allocated object. This is the second main reason for the use of a dynamic type system in this paper.

Dynamic flow sensitivity: There is a known limitation [21], [36] of the flow sensitivity of dynamic analyses. Consider the following example, where we assume h is 1 or 0 originally:

```
l = 1; t = 0; if (h == 1) t = 1; if (t != 1) l = 0;
```

If security labels are allowed to change freely under secret control, the above program results in the value of h being copied into l without changing the security label of l.

Our approach: In order to prevent such leaks it suffices to disallow security label upgrades under secret control. This corresponds to Austin and Flanagan's *no sensitive upgrade* [2] discipline. Hence, the execution of the above program is stopped with a security error.

Structure and existence: Not only the values of fields can be used to encode information, but also their presence or absence. Consider the following example:

```
if (h) o.q = 0;
```

After execution of the conditional the answer to the question whether q is in the domain of o gives away the value of h. It is important to note that not only the presence, but also the absence, of q gives away information — copying the value of h into l via the presence/absence of q can easily be done by executing `l = (o.q == undefined);`, since projecting non-existing fields returns undefined.

Our approach: We solve this by associating an *existence security label* with every field, and a *structure security label* with every object, demanding that addition of new fields under secret control is possible only if the structure of the object is secret. Once the structure of an object is secret, knowing the absence of fields in the object is also secret. In the example above $o.q$ would have secret existence security labels, and the structure of o would have to be secret for the body of the conditional to be allowed to be executed.

Permissiveness: As is, no sensitive upgrade prevents affecting public locations under secret control. This poses a permissiveness challenge: how to secure programs where public locations must be assigned under secret control.

Our approach: We address the challenge by facilitating upgrade of the security label before entering the secret control via the addition of a number of *upgrade* instructions to the language. All but one of the upgrade instructions is expressible in the language. Thus, the addition is merely one of convenience and does not extend the language.

Also note that the upgrade instructions are not necessary for the soundness of the system. They may be used to increase the permissiveness of the type system by communicating security policies to the runtime type checker. Although they can be used as part of the programming language, in order to support legacy code it is more fruitful to view it as a way to incorporate the results of an automatic analysis. We have successfully experimented with the use of automatic black-box testing to inject the upgrade instructions [7]. The idea is to run a modified version of the type system on automatically generated test cases, and, whenever the type system stops the execution, the trace of the execution is inspected to find the place in the program where an upgrade instruction must be inserted. The inserted upgrade instructions prevent the monitor from blocking the program and thus help avoiding false positives. Our experiments for a language with records and exceptions indicate that random testing accurately discovers annotations for a collection of scenarios with rich information flows.

Upgrade of the values of variables and fields: In order to write to variables or fields under secret control, the language provides instructions that upgrade the security label of the value of the variable or field. For instance, the example from the discussion on dynamic flow sensitivity above can be secured by adding appropriate upgrades:

```
l = 1; t = 0; upg t; if (h == 1) t = 1;
              upg l; if (t != 1) l = 0;
```

By upgrading the security label of the value of t before the conditional in which t is assigned we guarantee that the security label of the value of t is independent of secrets and compatible with the assignment (analogously for l). This renders a program that runs for all values of h, and where the results stored in variables t and l are always secret.

Upgrade of object structure and field existence: Addition and deletion of fields under secret control is only allowed on structurally secret records: the existence of the added field becomes secret, and only fields with a secret existence label are allowed to be removed. To enable fields to be added or deleted under secret control the structure security label of the record must be raised, as illustrated in the example below:

```
upg struct o; if (h) o.g = 1;
upg existence o.x; upg existence o.y;
if (h) delete o.x; else delete o.y;
```

Finally, it is not possible to express the upgrade of the existence security label of variables or of the structure security label of environment records (see Section III-B). Under some circumstances this hinders addition and deletion of variables under secret control. However, this restriction is in line with the design of JavaScript, where variable hoisting serves the purpose to make variable declarations independent of the control structure.

Exceptions: Exceptions offer further intricacies, since they allow for non-local control transfer. Any instructions following an instruction that may throw an exception based on a secret must be seen as being under secret control, since, similar to the conditional, the instruction may or may not be run based on whether the exception is thrown or not. Consider the following example:

```
l = true; if (h) {throw 0}; l = false;
```

Whether h is 0 or not controls whether the update is run or not and hence l encodes information about h.

Our approach: The default-allow policy for exceptions is clearly too restrictive — it would force us to treat all side effects after the first possible exception under secret control as secret. Instead, we introduce a dynamic security label for exceptions. If the *exception security label* is public, exceptions under secret control are prohibited. If, on the other hand, the security label is secret, any side effects are treated as if they were under secret control.

Let us return to the example above. Since the exception security label in the example is public, the conditional is prevented from throwing exceptions and the update of l can proceed without restrictions. If we want to allow the exception to be thrown, however, we upgrade the exception security label, in which case the update of l will be treated as being under secret control. On that account, the language provides an upgrade instruction $upg\ exc$ that upgrades the exception security label. Note that this instruction is not expressible in the language.

To make the example above run properly we must upgrade both the exception security label and the security label of the variable l.

```
l = true; upg l; upg exc;
if (h) {throw 0}; l = false;
```

Clearly, secret exceptions would be rather prohibitive if there was no way of lowering the exception security label. To this extent, the body of the *try-catch* statement can be used to encapsulate exception security label upgrades. Consider the following example:

```
l1 = true; upg l1;
try { upg exc; if (h) {throw 0}; l1 = false; }
catch (x) { skip }; l2 = 16;
```

In this example, since the upgrade of the exception security label occurs inside a *try-catch* statement , we know that regardless of whether an exception is thrown or not the assignment to l_2 will be run. Thus, the update of l_2 is public, whereas the update of l_1 remains secret. The exception handler is run under secret control given that the exception label is secret. The exception security label of the handler, however, is the initial exception security label, which means that exceptions are not allowed in the handler, unless exceptions under secret control are allowed by the outer exception security label.

Although the language must be extended with non-standard constructs to handle upgrading of the exception security label this is not a fundamental limitation. Under the assumption that normal execution is prevalent, i.e., that the exceptions are used only in cases where there is a failure. For such cases, the exception security label instruction can be provided in order to increase the permissiveness of the system. In the case the instruction is not made available only programs that must be able to recover secret exceptions are affected.

Higher-order functions: Being first class values, functions carry a security type that must be taken into consideration when calling the function. Consider the following example, where a secret function f is created by choosing between two functions under secret control.

```
f = if (h) {function g(x) { l = 1; }}
    else   {function g(x) { l = 0; }}
```

Depending on the secret h, a function that writes 1, or a functions that write 0 to l is chosen, and applying the result copies the value of h into l.

Our approach: The body of the function must be run in a security context that is at least as secret as the security label of the function. This discipline is inspired by static handling of higher-order functions [33].

Variables and scoping: JavaScript is known for its non-standard scoping rules. Variable hoisting in combination with non-syntactic scoping, the *with* and *eval* constructions, and the fact that updating non-existing variables will define the variable in the global variable environment, cause complex and sometimes unexpected behavior. To appreciate this consider, for instance, conditional shadowing using *eval*.

```
f = function f(x) {if (h) {eval("var l")}; l = 0}
```

In order to understand this example we must know more about functions and scoping in JavaScript. First, the variable environments form a chain. Variable lookup starts in the topmost environment and continues down the chain until the variable has been found or the end of the chain — the global variable environment — has been reached.

Second, both *if* and *eval* are unscoped, i.e., the variable declared by the *eval* is declared in the topmost variable environment of the function call. The use of eval is to prevent variable hoisting until the execution of the conditional branch. Otherwise, l would be hoisted out of the *if* and always be declared regardless of the value of h.

Execution of f is split into two cases. If h is true, the variable is defined locally in the function, and the update of l is captured by the local l. If h is false, l is created in the global variable environment (unless it already exists). Hence, both the value of l in the global variable environment and potentially its existence encodes information about h. The latter implies that we must track the structure of variable environments, similar to objects above. In addition, the presence or absence of the l in the local variable environment (or somewhere else in the chain in general) effects where the update takes place. This means that, when reading or writing to variables, the structure security labels encountered during the traversal of the variable chain must be taken into account.

Our approach: We show that the intricate variable behavior of JavaScript can be handled by a faithful modeling of the execution context in terms of objects.

C. From the core to full JavaScript

The core has been selected to be representative of the information flow challenges of JavaScript. The simplifications and omissions have been chosen to not exclude any information flow challenges. Hence, our approach is not a sub-language approach. Rather we envision that extension to the full language is possible without further technical development.

In addition, even though property attributes and other ECMA-262 (v.5) features like the ability to freeze objects and strict mode are important from a security perspective and might simplify enforcement of secure information flow we cannot assume that code is strict mode without sacrificing the possibility of handling legacy code. For this reason, we target the non-strict semantics of ECMA-262 (v.5).

With the intuition on the challenges and their resolution at hand, we are now ready to proceed to the formal development: the presentation of the dynamic type system and its soundness.

III. Language

The semantics is a subset of the non-strict semantics of ECMA-262 (v.5) standard. In order to aid the verification of the semantics and increase confidence in our modeling we have chosen to follow the standard closely.

The semantics is instrumented to track information flow with respect to a two-level security lattice, classifying information as either *public* or *secret*, preventing secret information from affecting public information. This is achieved by stopping the execution when potential leaks are detected, which is expressed in the semantics by not providing reduction rules for the cases that may cause insecure information flow. Stopping the execution when security violations are detected may introduce a one-bit information leak, which is reflected in our baseline security condition in Section IV.

$$
\begin{array}{lcl}
e & ::= & s \mid n \mid b \mid \mathit{undefined} \mid \mathit{null} \mid \mathit{this} \mid x \mid e_1[e_2] \mid e_1 = e_2 \\
 & \mid & \mathit{delete}\ e \mid \mathit{typeof}(e) \mid e_1 \star e_2 \mid e(\overline{e}) \mid \mathit{new}\ e(\overline{e}) \\
 & \mid & \mathit{function}\ x(\overline{x})\ c \\
c & ::= & \mathit{var}\ x \mid c_1; c_2 \mid \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2 \mid \mathit{while}\ (e)\ c \\
 & \mid & \mathit{for}\ (x\ \mathit{in}\ e)\ c \mid \mathit{throw}\ e \mid \mathit{try}\ c_1\ \mathit{catch}(x)\ c_2 \mid \mathit{return}\ e \\
 & \mid & \mathit{eval}\ e \mid \mathit{with}\ e\ c \mid \mathit{skip} \mid \mathit{upg}\ l \mid e \\
l & ::= & e \mid \mathit{existence}\ e \mid \mathit{struct}\ e \mid \mathit{exc}
\end{array}
$$

A. Syntax

Let $\overline{X}$ range over lists of X, where $\cdot$ denotes the cons operator and $[\,]$ denotes the empty list, e.g., $\overline{x}$ denotes lists of variable names, and $\overline{e}$ denotes lists of expressions.

The syntax of the language consists of two basic syntactic categories: expressions, ranged over by e, and statements, ranged over by c.

Unary operators are represented by *delete*, and *typeof*, and, without loss of generality, the binary operators are opaquely represented by $\star$. In addition, the empty statement is represented by a distinguished *skip* statement and the language is extended with nonstandard constructions for security label upgrade of variables and fields, existence security labels of fields, structure security labels of objects and the exception security label, explained in Section II-B. For simplicity we assume that each function contains exactly one return statement, and that it occurs as the last statement in the body of the function. Also, in the following, we identify string literals and variable names, writing s instead of "s".

B. Semantics

The semantics of the language is given in big-step operational form and provides dynamic security type checking.

This section consists of two parts. First, we introduce the values including a primitive notion of objects, the basis for the formulation of *ECMA objects*. This provides functionality for basic object interaction, extended to *function objects*, providing function call, and object construction via constructor functions, and *environment records*, providing the foundation for the lexical and variable environments. Second, we introduce the semantics of expressions and statements in terms of the development of the first part.

This stepwise construction allows for most of the information-flow control to be pushed into the basic primitives, simplifying the formulation of more complex functionality and their explanation.

$$
\begin{array}{lcl@{\qquad}lcl}
v & ::= & r \mid s \mid n \mid b \mid \mathit{undefined} & p & ::= & \dot{v} \mid \mathcal{F} \mid c \mid \overline{x} \\
o & ::= & \{s_1 \overset{\sigma_1}{\mapsto} p_1, \ldots, s_n \overset{\sigma_n}{\mapsto} p_n, \sigma\} & & & \\
\end{array}
$$

Values: Let H (secret) and L (public) be security labels, ranged over by σ, used to label the other values with security information. For clarity in the semantic rules, let pc and ϵ range over security labels in the function of *security context* and the exception security label, see Section II-B.

The primitive values, ranged over by v, are *primitive references* (i.e., opaque pointers), strings, numbers, booleans, and the *undefined* value. Similarly to security labels above, r ranges over primitive references in general, while le, ve, and τ range over primitive references when denoting *lexical environments*, *variable environments*, and the *this binding*, defined below.

In the instrumented semantics all primitive values occur together with a security label representing the security classification of the value. Let v^σ be security labeled primitive values, written $\dot{v}$ when the actual security label is unimportant. Let $\dot{v}^{\sigma_2} = v^{\sigma_1^{\sigma_2}} = v^{\sigma_1 \sqcup \sigma_2}$.

The *references*, $(\dot{r}, \dot{s})$, are pairs of security labeled primitive references and strings, denoting a field in an object. In the following we let $\dot{v}$ range over both security labeled value and references, $(\dot{r}, \dot{s})$.

In addition to the primitive values, the notion of primitive object, ranged over by o, forms the foundation of the semantics of JavaScript. A primitive object is a map from strings to *properties*, decorated with *existence* and *structure* security labels, see Section II-B. The fields are either *internal* or *external* indicated by the $IsExternal$ predicate on strings, which is false for strings of the form $\_s\_$ and true otherwise. Internal fields are used in the implementation of the semantics but are not exposed to the programmer — see the semantics of *for-in* below. The properties of external fields are security labeled primitive values, while internal fields may hold *algorithms* represented by general functions, $\mathcal{F}$, statements, c, and lists of formal parameters, $\overline{x}$.

The *heap*, ranged over by ϕ, is a mapping from primitive references to primitive objects. The *execution environment* $E ::= (\phi, \epsilon)$ is a pair of a heap, ϕ, and an exception security label, ϵ. Let $E[r] = \phi[r]$ for $E = (\phi, \epsilon)$. In addition, execution takes place with respect to an execution *context* $\mathcal{C} ::= (\dot{\tau}, \dot{le}, \dot{ve})$, built up by the *this* binding, τ, a primitive reference, le, to the topmost lexical environment, and a primitive reference, ve, to the topmost variable environment. Let $\dot{u}$ denote exception lifted values, i.e., $\dot{u} ::= \dot{v} \mid exc\ \dot{v}$.

ECMA Objects: All objects of JavaScript define a number of internal algorithm fields that provide a common interface for interacting with the object. Most of the standard is described in terms of this interface; only few algorithms manipulate the primitive objects directly. This common core provides an ideal location for handling information flow security. By showing that the core is secure we can easily establish (by using compositionality of our security notion) that more complex functionality formulated in terms of the core is secure as well.

The explanation of information flow in an object oriented language is facilitated by the notions of *read context*, and *write context*. The read context for an entity is the accumulated security label of the *access path* of the entity. For a field, the access path is the primitive reference to the object containing the field together with field name. When reading, the result is raised to security label of the read context.

The write context of an entity is the read context together with the security context and the exception security label. When writing to an entity the demand is that the security label of the entity is at least as secret as the write context of the entity. This ensures that the security labels are independent of secrets.

We define *ECMA Objects* to be primitive objects extended with a relevant selection of the core functionality. In particular, an ECMA Object $\mathbb{O}$ is defined by

$$\begin{aligned}\mathbb{O} = \{\ & \_GetOwnProperty\_ \mapsto GetOwnProperty,\\ & \_GetProperty\_ \mapsto GetProperty,\\ & \_HasProperty\_ \mapsto HasProperty,\\ & \_Delete\_ \mapsto Delete, \_Get\_ \mapsto Get, \_Put\_ \mapsto Put\ \}\end{aligned}$$

The rules for ECMA Objects are found in Table I.

GetOwnProperty: Given a primitive reference and field name GetOwnProperty returns a property descriptor, $\{value \mapsto \dot{v}\}$ containing the value of the field or *undefined* in the case the field does not exists. In both cases the security label of the result is raised to the read context (i.e. the primitive reference and field name). As discussed in Section II-B the structure security label is taken into account for non-existing fields.

GetProperty: Let d range over property descriptors or *undefined*, $d ::= \{value \mapsto \dot{v}\} | undefined^\sigma$, and let d^σ be defined structurally in the immediate way. GetProperty is formulated in terms of GetOwnProperty and follows the prototype chain [18] while searching for the field. During the search the security labels of the primitive references and of the GetOwnProperty results are accumulated and used to raise the final result, cf. the notion of read context and access path.

HasProperty: HasProperty is a boolean valued wrapper around GetProperty, i.e., it returns true if the field exists and false otherwise.

Delete: Delete deletes fields of objects. Given a primitive reference to a primitive object and a field name the field is removed from the object. Deleting an existing field from an object, changes the structure of the object, which causes the demand that the write context of the object is below the structure label, and similarly for the existence, see Section II-B. Similar to the update below, the structure security label is raised to the security label of the field name.

Get: Given a primitive reference and a field name Get uses GetOwnProperty to obtain a property descriptor and returns the value of the field or undefined if the field does not exist.

Put: Of the ECMA object algorithms Put is the most complicated from an information-flow perspective. It provides addition and update of the fields of objects, with different

$$\frac{o = E[r] \quad o = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots\}}{GetOwnProperty(r^{\sigma_1}, s^{\sigma_2})\ E = \{value \mapsto \dot{v}^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}\}} \qquad \frac{o = E[r] \quad s \notin dom(o) \quad o = \{\dots, \sigma_3\}}{GetOwnProperty(r^{\sigma_1}, s^{\sigma_2})\ E = undefined^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}}$$

$$\frac{GetOwnProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}}{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}} \qquad \frac{\begin{array}{c}GetOwnProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma_1} \\ o = E[r] \quad o[\_Prototype\_] = null^{\sigma_2}\end{array}}{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma_1 \sqcup \sigma_2}}$$

$$\frac{\begin{array}{c}GetOwnProperty(\dot{r}_1, \dot{s})\ E = undefined^{\sigma_1} \\ o = E[r_1] \quad o[\_Prototype\_] = \dot{r}_2 \quad GetProperty(\dot{r}_2, \dot{s})\ E = d\end{array}}{GetProperty(\dot{r}_1, \dot{s})\ E = d^{\sigma_1}} \qquad \frac{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto p^{\sigma}\}}{HasProperty(\dot{r}, \dot{s})\ E = true^{\sigma}}$$

$$\frac{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma}}{HasProperty(\dot{r}, \dot{s})\ E = false^{\sigma}} \qquad \frac{o = \phi[r] \quad o = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots, \sigma_4\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \quad \sigma <: \sigma_4 \quad \sigma \sqcup \sigma_2 <: \sigma_3}{pc \vdash Delete(r^{\sigma_1}, s^{\sigma_2})\ (\phi, \epsilon) = (true^L, (\phi[r \mapsto o \setminus s, \sigma_2 \sqcup \sigma_4], \epsilon))}$$

$$\frac{o = E[r] \quad s \notin dom(o)}{pc \vdash Delete(\dot{r}, \dot{s})\ E = (true^L, E)} \qquad \frac{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}}{Get(\dot{r}, \dot{s})\ E = \dot{v}} \qquad \frac{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma}}{Get(\dot{r}, \dot{s})\ E = undefined^{\sigma}}$$

$$\frac{\begin{array}{c}o_1 = \phi[r] \quad o_1 = \{s \overset{\sigma_3}{\mapsto} v_2^{\sigma_4}, \dots, \sigma_5\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \\ \sigma \sqcap \sigma_2 <: \sigma_5 \quad \sigma \sqcup \sigma_2 <: \sigma_4 \quad o_2 = o_1[s \overset{(\sigma \sqcup \sigma_2) \sqcap \sigma_3}{\mapsto} \dot{v}_1^{\sigma \sqcup \sigma_2}, \sigma_2 \sqcup \sigma_5]\end{array}}{pc \vdash Put(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)} \qquad \frac{\begin{array}{c}o_1 = \phi[r] \quad s \notin dom(o_1) \quad o_1 = \{\dots, \sigma_3\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \\ \sigma <: \sigma_3 \quad o_2 = o_1[s \overset{\sigma \sqcup \sigma_2}{\mapsto} \dot{v}_1^{\sigma \sqcup \sigma_2}, \sigma_2 \sqcup \sigma_3]\end{array}}{pc \vdash Put(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

Table I
ECMA OBJECTS

information-flow restrictions depending on whether the field is already present or not.

In the case of addition, the structure of the object is changed, and the demand is that the previous structure security label is at least as secret as the write context of the object. In addition, flow sensitivity allows the structure security label to be raised to the security label of the field name. Thus, adding field with a secret field name under public control to an object with public structure will make the structure of the object secret. The existence security label and the value security label are both raised to their write contexts.

In the case of update, the structure of the object is not changed, but the absence of change encodes information about the field name. If both the write context of the object and the security label of the field name are secret it is demanded that the structure of the object is secret. Otherwise, the structure is raised to the security label of the field name. The existence security label is either retained or lowered, to allow for the existence security label to become public if the field is updated in a public write context using a public field name. Finally, the value security label is raise to the write context of the field value.

Let $NewEcma$ allocate a new ECMA Object and return the primitive reference to the newly allocated object

$$\frac{r\ fresh \quad \phi_2 = \phi_1[r \mapsto \mathbb{O}[pc \sqcup \epsilon]]}{pc \vdash NewEcma\ (\phi_1, \epsilon) = (r^L, (\phi_2, \epsilon))}$$

where $o[\sigma]$ denotes updating the structure security label.

For convenience, we use the standard dot notation for method application defined as follows, $r^{\sigma}.X(\overline{a})\ E = (E[r][X](r^{\sigma} \cdot \overline{a})\ E)^{\sigma}$, where $\overline{a}$ (the arguments) denotes a list of security labeled values, $\dot{v}$.

Initial environment and built-in objects: In addition to the functionality provided by the expressions and statements of the language the standard execution environment contains a number of built-in objects reachable from the *global object*.

In the following, public existence security labels and value security labels of internal fields are frequently omitted in the case the field is mandatory or cannot be updated.

We define a minimal initial environment, $\phi_{init} = \{\ r_G \mapsto \mathbb{G}, r_O \mapsto Object\ \}$, containing only the global object, $\mathbb{G}$, and the object constructor, $Object$, defined below.

Global object: In JavaScript, the global object defines a number of fields that enriches the execution environment with constants, functions and constructors. In addition to this, the global object acts as the store for global variables. For space reasons, we include only the object constructor, $Object$, as a constructor property of the global object, refraining from including the other constructors and prototypes.

$$\mathbb{G} = \mathbb{O}[\ \_Prototype\_ \mapsto null,\ Object \mapsto r_O\ ]$$

Object constructor: The object constructor, $Object = \mathbb{O}[\ \_Prototype\_ \mapsto null, prototype \mapsto null, \_Construct\_ \mapsto ObjectConstruct\ ]$, is a function object that provides functionality for object creation, via the internal $\_Construct\_$ field. See below for more information on function objects. The ObjectConstruct algorithm allocates a new object, initializes the $\_Prototype\_$ field to $null$, and returns the new primitive reference.

$$\frac{\begin{array}{c}(\dot{r}_2, E_2) = pc \vdash NewEcma\ E_1 \\ E_3 = pc \vdash \dot{r}_2.\_Put\_(\_Prototype\_^L, null^L)\ E_2\end{array}}{pc \vdash ObjectConstruct(\dot{r}_1, [\ ])\ E_1 = (\dot{r}_2, E_3)}$$

Variable environment and environment records: The variable environment is a chain of environment

records, chained together by *chaining objects*, $\mathbb{C}(r_1, \dot{r}_2) = \{\_EnvironmentRecord\_ \mapsto r_1, \_OuterEnvironment\_ \mapsto \dot{r}_2\}$, where *EnvironmentRecord* points to the environment record, and *OuterEnvironment* points to the next chaining object in the chain. The environment records store the variable bindings and come in two forms: *object environment records* and *declarative environment records* differing in whether a separate object, the *binding object*, is used to store the variable bindings or if the bindings are stored in the environment record itself. Another difference is in the implementation of the *ImplicitThisValue*, for which the object environment record returns the binding object and the declarative environment record returns *null*.

We simplify object environment records and declarative environment records to support the same subset of operations: *HasBinding*, *GetBindingValue*, *SetMutableBinding*, *DeleteBinding*, and *ImplicitThisValue*.

The $NewDeclarativeEnvironment$ algorithm allocates a new declarative environment and links it onto the given environment record chain, and similarly for the $NewObjectEnvironment$ algorithm for object environments.

$$\frac{r_1, r_2 \ fresh \qquad E_2 = E_1[r_1 \mapsto \mathbb{DE}, r_2 \mapsto \mathbb{C}(r_1, \dot{r})]}{pc \vdash NewDeclarativeEnvironment(\dot{r}) \ E_1 = (r_2^L, E_2)}$$

$$\frac{r_3, r_4 \ fresh \qquad E_2 = E_1[r_3 \mapsto \mathbb{OE}(\dot{r}_2), r_4 \mapsto \mathbb{C}(r_3, \dot{r}_1)]}{pc \vdash NewObjectEnvironment(\dot{r}_1, \dot{r}_2) \ E_1 = (r_4^L, E_2)}$$

Get identifier reference: GetIdentifierReference takes a primitive reference to the topmost variable environment and a variable name and traverses the chain of environment records until the variable is found or the chain ends. The returned result is a reference $(\dot{r}, x^L)$ where x is the name of the variable and r is a primitive reference to the environment record containing the variable, or $undefined$ if the variable was not found.

$$\frac{}{GetIdentifierReference(null^\sigma, x) \ E = (undefined^\sigma, x^L)}$$

$$\frac{\begin{array}{c} r_1 \neq null \qquad r_2 = E[r_1][\_EnvironmentRecord\_] \\ true^{\sigma_2} = r_2^{\sigma_1}.\_HasBinding\_(x^L) \ E \end{array}}{GetIdentifierReference(r_1^{\sigma_1}, x) \ E = (r_2^{\sigma_2}, x^L)}$$

$$\frac{\begin{array}{c} r_1 \neq null \qquad r_2 = E[r_1][\_EnvironmentRecord\_] \\ false^{\sigma_2} = r_2^{\sigma_1}.\_HasBinding\_(x^L) \ E \\ \dot{r}_3 = E[r_1][\_OuterEnvironment\_] \\ (\dot{r}_4, \dot{x}) = GetIdentifierReference(\dot{r}_3^{\sigma_2}, x) \ E \end{array}}{GetIdentifierReference(r_1^{\sigma_1}, x) \ E = (\dot{r}_4, \dot{x})}$$

Function objects: Function objects are objects containing two internal properties, $\_Call\_$, used by function application and $\_Construct\_$, used in object construction.

User defined function objects: The function objects created when evaluating function expressions are closures storing the context the function was created in, the formal parameters of the function and the code of the function, used by associated $\_Call\_$, and $\_Construct\_$. Let $\mathbb{F}$ denote the family of function objects

$$\begin{array}{l} \mathbb{F}(\overline{x}, c, \dot{r}) = \mathbb{O}[\ \_Scope\_ \mapsto \dot{r}, \_FormalParameters\_ \mapsto \overline{x}, \\ \qquad\qquad \_Code\_ \mapsto c, \_Call\_ \mapsto FunctionCall, \\ \qquad\qquad \_Construct\_ \mapsto FunctionConstruct\ ] \end{array}$$

Call: Calling a user defined function first allocates a new declarative environment in which the arguments are bound by a process called declaration binding. Thereafter the body of the function is executed in the updated context. Note that the creation of the declarative environment, the declaration binding, and the body are run in a security context raised to the security label of the primitive reference, r_F, of the function, as is the returned value, see Section II-B.

$$\frac{\begin{array}{c} \mathbb{F} = E_1[r_F] \qquad\qquad pc_2 = pc_1 \sqcup \sigma \\ (\dot{r}, E_2) = pc_2 \vdash NewDeclarativeEnvironment(\mathbb{F}[\_Scope\_]^\sigma) \ E_1 \\ E_3 = pc_2, \dot{r} \vdash DeclarationBinding(\mathbb{F}, \overline{a}) \ E_2 \\ pc_2, (this(\dot{r}_t), \dot{r}, \dot{r}) \vdash \langle \mathbb{F}[\_Code\_], E_3 \rangle \rightarrow \langle \dot{u}, E_4 \rangle \end{array}}{pc_1 \vdash FunctionCall(\dot{r}_t, r_F^\sigma, \overline{a}) \ E_1 = (\dot{u}^\sigma, E_4)}$$

where $this(undefined^\sigma) = r_G^\sigma$, and $this(\dot{r}) = \dot{r}$, otherwise. (Recall that r_G is the primitive reference to the global object defined above.) See below for the definition the semantics of statements used to evaluate the body, $\mathbb{F}[\_Code\_]$.

Declaration binding: Declaration binding first binds the arguments of the function call and then performs variable hoisting.

$$\frac{\begin{array}{c} E_2 = pc, \dot{r} \vdash BindParameters(\mathbb{F}[\_FormalParameters\_], \overline{a}) \ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(\mathbb{F}[\_Code\_]) \ E_2 \end{array}}{pc, \dot{r} \vdash DeclarationBinding(\mathbb{F}, \overline{a}) \ E_1 = E_3}$$

The parameters are bound by ignoring any surplus parameters, and setting missing parameters to $undefined$.

$$pc, \dot{r} \vdash BindParameters([\,], \_) \ E = E$$

$$\frac{\begin{array}{c} E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, \dot{v}) \ E_1 \\ E_3 = pc, \dot{r} \vdash BindParameters(\overline{x}, \overline{v}) \ E_2 \end{array}}{pc, \dot{r} \vdash BindParameters(x \cdot \overline{x}, \dot{v} \cdot \overline{v}) \ E_1 = E_3}$$

$$\frac{\begin{array}{c} E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, undefined^L) \ E_1 \\ E_3 = pc, \dot{r} \vdash BindParameters(\overline{x}, [\,]) \ E_2 \end{array}}{pc, \dot{r} \vdash BindParameters(x \cdot \overline{x}, [\,]) \ E_1 = E_3}$$

Variable hoisting traverses the body of the function and defines all variables, initializing them to $undefined$. The hoisting algorithm $HoistVariables$ is easily formulated and can be found in the full version of the paper [26].

Construct: Object construction via user defined functions uses the function as an initializer on a newly allocated ECMA object. Before passing the object, the $\_Prototype\_$ field is set to the value of the $prototype$ field of the constructor function, after which the object is initialized by calling the constructor function using the primitive reference

to the object as the *this* argument. Without loss of generality, we assume that all constructor functions return a primitive reference to the new object.

$$\frac{\begin{array}{c}(\dot{r}_1, E_2) = NewEcma\ E_1 \qquad pc_2 = pc_1 \sqcup \sigma \\ \dot{r}_2 = r_F^\sigma.\_Get\_(prototype)\ E_2 \\ E_3 = pc_2 \vdash \dot{r}_1.\_Put\_(\_Prototype\_^L, \dot{r}_2)\ E_2 \\ (\dot{u}, E_4) = pc_2 \vdash E_3[r_F][\_Call\_](\dot{r}_1, r_F^\sigma, \overline{a})\ E_3\end{array}}{pc_1 \vdash FunctionConstruct(r_F^\sigma, \overline{a})\ E_1 = (\dot{u}^\sigma, E_4)}$$

In the following let $NewFun$ allocate and set up a new function object from the given list of formal parameters, function body and scope.

$$\frac{\begin{array}{c}r_1\ fresh \quad E_2 = E_1[r_1 \mapsto \mathbb{F}(\overline{x}, c, \dot{r})] \quad (\dot{r}_2, E_3) = NewEcma\ E_2 \\ E_4 = pc \vdash \dot{r}_2.\_Put\_(constructor^L, r_1^L)\ E_3 \\ E_5 = pc \vdash r_1^L.\_Put\_(prototype^L, \dot{r}_2)\ E_4\end{array}}{pc \vdash NewFun(\overline{x}, c, \dot{r})\ E_1 = (r_1^L, E_5)}$$

Auxiliary reference functions: GetValue fetches the value associated with a reference; non-reference values are returned untouched. Subject to the limitations of Section II-B an exception is raised if the reference is undefined.

$$\frac{}{pc \vdash GetValue(v^\sigma)\ E = v^\sigma}$$

$$\frac{\dot{v} = \dot{r}.\_Get\_(\dot{s})\ E \qquad r \neq undefined}{pc \vdash GetValue((\dot{r}, \dot{s}))\ E = \dot{v}}$$

$$\frac{\sigma \sqcup pc <: \epsilon \qquad r = undefined}{pc \vdash GetValue((r^\sigma, \dot{s}))\ (\phi, \epsilon) = exc\ ReferenceError^\sigma}$$

PutValue takes a reference and a value and updates the location denoted by the reference with the value — if the reference is undefined the update is done on the global object. Subject to the limitations of Section II-B an exception is raised if the first argument is not a reference.

$$\frac{\sigma \sqcup pc <: \epsilon}{pc \vdash PutValue(v_1^\sigma, \dot{v}_2)\ (\phi, \epsilon) = (exc\ ReferenceError^\sigma, (\phi, \epsilon))}$$

$$\frac{E_2 = pc \vdash p_G^\sigma.\_Put\_(\dot{s}, \dot{v})\ E_1}{pc \vdash PutValue((undefined^\sigma, \dot{s}), \dot{v})\ E_1 = E_2}$$

$$\frac{r \neq undefined \qquad pc \vdash \dot{r}.\_Put\_(\dot{s}, \dot{v})\ E_1 = E_2}{pc \vdash PutValue((\dot{r}, \dot{s}), \dot{v})\ E_1 = E_2}$$

Semantics of expressions: The semantics of expressions is of the form $pc, \mathcal{C} \vdash \langle e, E_1 \rangle \rightarrow \langle \dot{u}, E_2 \rangle$, read as e reduces to $\langle \dot{u}, E_2 \rangle$ when run in E_1, the security context pc, and the context $\mathcal{C}$.

The semantic rules for expressions are found in Table II, where the exception propagation rules have been omitted for clarity. We refer the reader to the full version of the paper [26] for the remaining rules.

The expression rules are fairly straightforward, and formulated in terms of the primitives of the previous section. Identifier dereference uses $GetIdentifierReference$. All rules that contain evaluation of subexpressions use $GetValue$ to convert the results to values, assignment uses $PutValue$ to update the location denoted by the reference. ***delete*** uses $Delete$ or $DeleteBinding$ depending on if the target is an object or an environment record (indicated by $IsPropertyReference$). Function creation uses $NewDeclarativeEnvironment$ to allocate the local variable environment, $NewFun$ to create the new function object and $SetMutableBinding$ to initialize the new environment by binding the function name to the newly created function object in order to allow for recursive calls. Function application uses $Call$, and object creation uses $Construct$.

The field projection, assignment, object construction, and binary operators all use the evaluation of a list of expressions $pc, \mathcal{C} \vdash \langle \overline{e}, E_1 \rangle \rightarrow \langle \overline{v}, E_2 \rangle$, which returns a list of security labeled values.

Note the $this(\dot{v})\ E_3$ of the function call, which computes the *this* binding of the invocation, from the reference $\dot{v} = (r, s)^\sigma$: 1) if the base of the reference, r, is undefined, or if it is a property reference then the base of the reference is returned, otherwise, 2) the base is an environment record and the result of calling $\_ImplicitThisValue\_$ is returned.

Finally, note how *typeof* returns *undefined* for undefined variables, whereas using an undefined variable in, e.g., a binary operator would cause an exception when trying to apply $GetValue$ on the reference. This is the reason *typeof* can be used to detect whether variables are defined or not.

From an information flow perspective only the rules for *typeof* and binary operators contain primitive information flow (corresponding to the standard treatment of unary and binary operators); the information flow of the rest of the rules is in terms of primitive constructions.

Semantics of statements: Let $R ::= E \mid exc\ E \mid \langle \dot{u}, E \rangle$ indicating normal termination without return value, exceptional termination, and termination with return value $\dot{u}$. The semantics of statements is of the form $pc, \mathcal{C} \vdash \langle c, E \rangle \rightarrow R$, read as c reduces to R when run in E, security context pc and context $\mathcal{C}$.

The semantic rules of statements are found in Table III, where the rules for exception propagation have been omitted for clarity. We refer the reader to the full version of the paper [26] for the remaining rules.

Sequence, iteration in the form of *while*, and conditional choice are all standard. The two latter raise the security context of the body and chosen branch to the security label of the controlling expression.

The *for-in* statement iterates over the external fields of an object. For each external field name in the object, the given variable is updated with the name using the existence security label of the field as the security label, and the body of the *for-in* is run. The iteration is provided by the three rules of the form $pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_1 \rangle \rightarrow E_2$, that for each $\dot{s}$ in $\overline{s}$ binds the variable references by $\dot{v}$ to $\dot{s}$ before running c. Recall that $\overline{s}$ ranges over lists of security labeled strings.

$$pc, \mathcal{C} \vdash \langle s|n|b|undefined, E\rangle \to \langle s^L|n^L|b^L|undefined^L, E\rangle$$
$$pc, \mathcal{C} \vdash \langle null, E\rangle \to \langle 0^L, E\rangle \quad pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle this, E\rangle \to \langle \dot{\tau}, E\rangle$$

$$pc, \mathcal{C} \vdash \langle [\,], E\rangle \to \langle [\,], E\rangle$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}_1, E_2\rangle \quad \dot{v}_2 = pc \vdash GetValue(\dot{v}_1)\ E_2 \quad pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \to \langle \overline{a}, E_3\rangle}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \to \langle \dot{v}_2 \cdot \overline{a}, E_3\rangle}$$

$$\frac{\dot{v} = GetIdentifierReferece(\dot{le}, x)\ E}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle x, E\rangle \to \langle \dot{v}, E\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \quad \dot{r} = pc \vdash GetValue(\dot{v}_1)\ E_2 \quad \dot{s} = pc \vdash GetValue(\dot{v}_2)\ E_2}{pc, \mathcal{C} \vdash \langle e_1[e_2], E_1\rangle \to \langle (\dot{r}, \dot{s}), E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \quad \dot{v}_3 = pc \vdash GetValue(\dot{v}_2)\ E_2 \quad E_3 = pc \vdash PutValue(\dot{v}_1, \dot{v}_3)\ E_2}{pc, \mathcal{C} \vdash \langle e_1 = e_2, E_1\rangle \to \langle \dot{v}_3, E_3\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle (\dot{r}, \dot{s}), E_2\rangle \quad IsPropertyReference((r, s))\ E_2 \quad (\dot{v}, E_3) = pc \vdash \dot{r}.\_Delete\_(\dot{s})\ E_2}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \to \langle \dot{v}, E_3\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle (undefined^\sigma, \dot{s}), E_2\rangle}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \to \langle true^L, E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle (\dot{r}, \dot{s}), E_2\rangle \quad \neg IsPropertyReference((r, s))\ E_2 \quad (\dot{v}, E_3) = pc \vdash \dot{r}.\_DeleteBinding\_(\dot{s})\ E_2}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \to \langle \dot{v}, E_3\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}, E_2\rangle \quad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \quad pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \to \langle \overline{a}, E_3\rangle \quad (\dot{u}, E_4) = pc \vdash E_3[r][\_Call\_](this(\dot{v})\ E_3, \dot{r}, \overline{a})\ E_3}{pc, \mathcal{C} \vdash \langle e(\overline{e}), E_1\rangle \to \langle \dot{u}, E_4\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}, E_2\rangle \quad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \quad pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \to \langle \overline{a}, E_3\rangle \quad (\dot{u}, E_4) = pc \vdash E_3[r][\_Construct\_](\dot{r}, \overline{a})\ E_3}{pc, \mathcal{C} \vdash \langle new\ e(\overline{e}), E_1\rangle \to \langle \dot{u}, E_4\rangle}$$

$$\frac{(le_2^\sigma, E_2) = pc \vdash NewDeclarativeEnvironment(\dot{le}_1)\ E_1 \quad (\dot{r}_f, E_3) = pc \vdash NewFun(\overline{x}, c, le_2^\sigma)\ E_2 \quad \dot{r} = E_3[le_2][\_EnvironmentRecord\_] \quad E_4 = \dot{r}^\sigma.\_SetMutableBinding\_(x^L, \dot{r}_f)\ E_3}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle function\ x(\overline{x})\ c, E_1\rangle \to \langle \dot{r}_f, E_4\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \quad v_3^{\sigma_1} = pc \vdash GetValue(\dot{v}_1)\ E_2 \quad v_4^{\sigma_2} = pc \vdash GetValue(\dot{v}_2)\ E_2}{pc, \mathcal{C} \vdash \langle e_1 \star e_2, E_1\rangle \to \langle (v_3 \star v_4)^{\sigma_1 \sqcup \sigma_2}, E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle v^\sigma, E_2\rangle}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \to \langle typeof(v)^\sigma, E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle (\dot{r}, \dot{s}), E_2\rangle \quad r \neq undefined \quad v^\sigma = GetValue((\dot{r}, \dot{s}))\ E_2}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \to \langle typeof(v)^\sigma, E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle (undefined^\sigma, \dot{s}), E_2\rangle}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \to \langle undefined^\sigma, E_2\rangle}$$

Table II
SEMANTICS OF EXPRESSIONS

$$\frac{pc, \mathcal{C} \vdash \langle c_1, E_1\rangle \to E_2 \quad pc, \mathcal{C} \vdash \langle c_2, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle c_1; c_2, E_1\rangle \to E_3}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle true^\sigma, E_2\rangle \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c_1, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle if\ (e)\ c_1\ else\ c_2, E_1\rangle \to E_3}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle false^\sigma, E_2\rangle \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c_2, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle if\ (e)\ c_1\ else\ c_2, E_1\rangle \to E_3}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle false^\sigma, E_2\rangle}{pc, \mathcal{C} \vdash \langle while\ (e)\ c, E_1\rangle \to E_2}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle true^\sigma, E_2\rangle \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c, E_2\rangle \to E_3 \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle while\ (e)\ c, E_3\rangle \to E_4}{pc, \mathcal{C} \vdash \langle while\ (e)\ c, E_1\rangle \to E_4}$$

$$pc, \mathcal{C} \vdash \langle ([\,], \dot{v}, c), E\rangle \to \langle E\rangle$$

$$\frac{IsExternal(s) \quad E_2 = pc \vdash PutValue(\dot{v}, s^\sigma)\ E_1 \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c, E_2\rangle \to E_3 \quad pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_3\rangle \to E_4}{pc, \mathcal{C} \vdash \langle (s^\sigma \cdot \overline{s}, \dot{v}, c), E_1\rangle \to E_4}$$

$$\frac{\neg IsExternal(s) \quad pc, \mathcal{C} \vdash \langle (\overline{s}, \dot{v}, c), E_1\rangle \to E_2}{pc, \mathcal{C} \vdash \langle (s^\sigma \cdot \overline{s}, \dot{v}, c), E_1\rangle \to E_2}$$

$$\frac{pc <: \epsilon}{pc, \mathcal{C} \vdash \langle upg\ exc, (\phi, \epsilon)\rangle \to (\phi, H)}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{u}, E_2\rangle}{pc, \mathcal{C} \vdash \langle return\ e, E_1\rangle \to \langle \dot{u}, E_2\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle x, E_1\rangle \to \langle \dot{v}_1, E_1\rangle \quad pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}_2, E_2\rangle \quad r^\sigma = pc \vdash GetValue(\dot{v}_2)\ E_2 \quad E_2[r] = \{s_1 \overset{\sigma_1}{\mapsto} \dot{p}_1, \ldots, s_n \overset{\sigma_n}{\mapsto} \dot{p}_n, \ldots\} \quad pc, \mathcal{C} \vdash \langle ([s_1^{\sigma \sqcup \sigma_1}, \ldots, s_n^{\sigma \sqcup \sigma_n}], \dot{v}_1, c), E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle for\ (x\ in\ e)\ c, E_1\rangle \to E_3}$$

$$\frac{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle c_1, (\phi_1, \epsilon_1)\rangle \to \langle exc\ \dot{v}, E_2\rangle \quad (le_2^\sigma, E_3) = pc \vdash NewDeclarativeEnvironment(\dot{le}_1)\ E_2 \quad \dot{r} = E_3[le_2][\_EnvironmentRecord\_] \quad (\phi_4, \epsilon_4) = \dot{r}^\sigma.\_SetMutableBinding\_(x^L, \dot{v})\ E_3 \quad pc \sqcup \epsilon_4, (\dot{\tau}, le_2^\sigma, \dot{ve}) \vdash \langle c_2, (\phi_4, \epsilon_1)\rangle \to E_5}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1)\rangle \to E_5}$$

$$\frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}, (\phi_2, \epsilon_2)\rangle \quad pc <: \epsilon_2}{pc, \mathcal{C} \vdash \langle throw\ e, E_1\rangle \to \langle exc\ \dot{v}, (\phi_2, \epsilon_2)\rangle}$$

$$\frac{pc, \mathcal{C} \vdash \langle c_1, (\phi_1, \epsilon_1)\rangle \to (\phi_2, \epsilon_2)}{pc, \mathcal{C} \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1)\rangle \to (\phi_2, \epsilon_1)}$$

$$\frac{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle e, E_1\rangle \to \langle s^\sigma, E_2\rangle \quad c = parse(s) \quad E_3 = pc, \dot{ve} \vdash HoistVariables(c)\ E_2 \quad pc \sqcup \sigma, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle c, E_2\rangle \to E_3}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle eval\ e, E_1\rangle \to E_3}$$

$$\frac{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle e, E_1\rangle \to \langle \dot{v}, E_2\rangle \quad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \quad (\dot{le}_2, E_3) = pc \vdash NewObjectEnvironment(\dot{le}_1, \dot{r})\ E_2 \quad pc, (\dot{\tau}, \dot{le}_2, \dot{ve}) \vdash \langle c, E_3\rangle \to E_4}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle with\ e\ c, E_1\rangle \to E_4}$$

Table III
SEMANTICS OF STATEMENTS

The *throw* demands that the security context pc is below the exception security label ϵ, as does the exception security label upgrade statement.

The execution of *try-catch* is divided into normal and exceptional execution of the body. In the first case, the result of the execution is returned in the *outer exception context*, i.e., the exception security label of the *try-catch*. This allows for containing exception security label upgrades to the *try-catch*. In the second case, if an exception is thrown in the body of the *try-catch*, control is passed to the exception handler. The body of the handler is run in a new lexical environment, in which the formal parameter of the handler is bound to the exception value. This means that variables declared in the body of the handler are not contained to the body of the handler, similar to *eval* and *with* above. With respect to information flow, the body of the handler is run in a security context which is the least upper bound of the initial security context and the exception security label at the program point where the exception was thrown. This guarantees that the body of the handler is unable to leak information about the existence of secret exceptions. Further, the body of the handler is run in the outer exception security level, since any (uncaught) exceptions in the handler escape the *try-catch*.

The *eval* statement evaluates its argument, parses the result to a program, which is run after hoisting the variables into variable environment. Hence, variables introduced by *eval* are defined in the context of the closest enclosing function, or into the global object. The program is run in a security context that is raised to the security label of the parsed string.

Finally, the *with* statement changes the lexical environment, hence shadowing existing variables, but not containing any potential variable declarations contained in the body of the *with*, since the variable hoisting declares the variable in the variable environment.

IV. Information-flow security and Transparency

A common policy for information-flow security is noninterference [11], [22]. Formally, noninterference is formulated as the preservation of a family of *low-equivalence* relations, $\sim$, under execution. As is standard for languages with references [5], the family is indexed by a relation β representing a bijection between the public domains of the heaps.

There are several flavors of noninterference depending on whether timing, progress, and termination are taken into account. In the following, we consider a baseline policy of *termination-insensitive* [44], [39] noninterference: two expressions are considered *noninterfering* if all terminating runs agree on public outcomes. Termination-insensitive noninterference allows leaks of information via the termination behavior of programs. In a batch-job setting, it allows leaks of at most one bit of information. Termination-insensitive noninterference is a natural fit for the monitor because it justifies the blocking upon detecting a security violation.

A. Low-equivalence

Noninterference is formulated in terms of a family of low-equivalence relations $\cdot \sim_{\cdot} \cdot$ for values, objects, heaps, and environments. The family of relations is defined structurally, demanding that equivalent values carry equal security labels, and in the case the label is public that values are equal. The definition of the low-equivalence relation can be found in the full version of the paper [26].

B. Noninterference

Two statements c_1 and c_2 are noninterfering, $ni(c_1, c_2)$, if any pair of terminating runs, starting from low-equivalent execution environments, results in low-equivalent execution environments:

$$\begin{aligned} ni(c_1,c_2) = E_1 \sim_{\beta_1,\epsilon_1} E_2 \wedge \mathcal{C}_1 \sim_{\beta_1} \mathcal{C}_2 \wedge \\ L, \mathcal{C}_1 \vdash \langle c_1, E_1\rangle \rightarrow \langle \dot{u}_1, E_1'\rangle \wedge L, \mathcal{C}_2 \vdash \langle c_2, E_2\rangle \rightarrow \langle \dot{u}_2, E_2'\rangle \Rightarrow \\ \exists \beta_2, \epsilon_2 \;.\; \beta_1 \subseteq \beta_1 \wedge \langle \dot{u}_1, E_1'\rangle \sim_{\beta_2,\epsilon_2} \langle \dot{u}_2, E_2'\rangle \end{aligned}$$

We prove the security of the dynamic type system by establishing that all terminating runs of all programs are noninterfering:

Theorem 1 (Noninterference)**.** $\forall c \;.\; ni(c, c)$.

Proof: The proof (detailed in the full version of this paper [26]) proceeds by induction on the size of the execution derivation tree. As is standard, the noninterference theorem is uses a supporting lemma that proves freedom of public side effects under secret control. The proof includes a number of tricky sub-proofs — in particular *Put*, and *try-catch* and parts of the exception propagation. Those sub-proofs have been independently stated as lemmas, and have been formalized and proved using the proof assistant Coq. ■

C. Transparency

Transparency expresses that the security instrumentation is conservative, i.e., if a program is able to run in the instrumented semantics, then this run is consistent with the run of the program in the original (un-instrumented) semantics. Let $\rightsquigarrow$ denote evaluation in the un-instrumented semantics that ignores security labels and interpret upgrade instructions as *skip*. Let Φ be a function that removes all security labels from values.

Theorem 2 (Transparency)**.** *It holds that*

$$L, \mathcal{C} \vdash \langle c, E_1\rangle \rightarrow \langle \dot{u}, E_2\rangle \Rightarrow \Phi(\mathcal{C}) \vdash \langle c, \Phi(E_1)\rangle \rightsquigarrow \langle \Phi(\dot{u}), \Phi(E_2)\rangle$$

V. Scenarios

This section presents the implementation of the two scenarios of Section III.

A. Online advertisement

Consider an online shopping cart holding a number of items. The cart contains a number of items, displaying their name and their prices, and the total price. In addition, the merchant wants to include ads from cooperating third parties and discounts mixed in with the items to get the information as close to the relevant item.

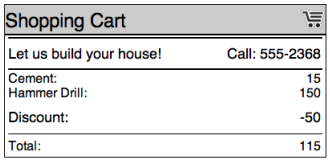


Figure 1. Shopping Cart Example

The resulting situation is that items, discounts and third party code are all mixed, when rendering the cart. This gives rise to a number of interesting information-flow issues. For instance, the customer may want to keep the existence of certain items, their prices, or the number of items secret. Similarly, the merchant may want to keep any discounts or offers secret, as they reflect a particular customer relationship.

In order to construct the above example we have implemented the core structure of DOM nodes, faithfully implementing the interface mandated by the DOM [27]. We allow the programmer full access to the link structure in terms of *parent*, *firstChild*, *lastChild*, *prevSibling*, *nextSibling*, and the *childNodes* array. Insertion and removal of nodes is provided via *insertBefore*, *appendChild*, *replaceChild*, and *removeChild*. As mandated by the recommendations [27], repeated insertion of the same node will move the node in the structure.

The cart is implemented using the structure of an unordered list. This produces a linked structure mixing public and secret data, and where the existence of certain elements might be secret, depending on the security policies of the merchant and the customer. Assume that the nodes of the example ul, A, $I1$, $I2$, D, and T have been created and properly initialized. The names should be interpreted as follows: ul stands for unordered list, I for item, A for ad, D for discount and T for total.

Assuming that the customer wants to protect the price of the items and the total price, but not the items themselves, and that the merchant wants to protect the existence of discount, we can build the list of the cart by successive use of *appendChild* on ul.

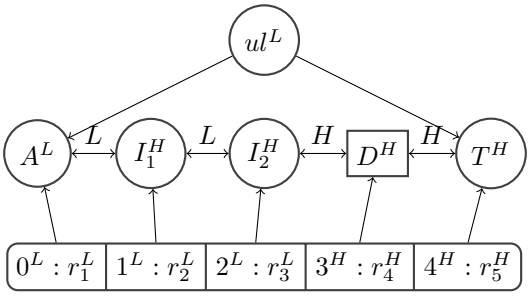
```
ul.appendChild(A); ul.appendChild(I1);
ul.appendChild(I2); ...
```

This code adds the ad, and the first items to the list. Thereafter, the discount is added under secret control using the expression **if** (h) top.appendChild(D);, where $h = 1^H$. This addition is not allowed without changing the security labels of the existing structure, since it is being written into under secret control. The following sequence of label upgrades covers the footprint of the *appendChild* method.

```
upg_struct(ul.childNodes);
upg(ul.childNodes.length);
upg(ul.lastChild);
upg(I2.nextSibling); upg(D.nextSibling);
upg(D.prevSibling); upg(D.parent);
```

As discussed in Section II such upgrade sequences can be generated by automatic testing. Finally, we add the last node, the total, top.appendChild(T);.

With some of the edges suppressed for clarity, the result of the program is the node hierarchy depicted below. Note how the contents of I_1, I_2, D, and T are secret, and how the existence of the discount D is secret, indicated by the secret references between D and its neighbors. In the bottom is the child nodes array, where the existence of the first three indices is public, and the existence of the last two is secret, i.e. there exists a point in the array dividing the array into a public left part and a secret right part. The type of this array corresponds to the structural restrictions imposed by Russo et al. [37]. It is interesting to note how this demand arises naturally in the implementation without imposing any restrictions, but rather based on how the array is used by the node insertion code. Note that this applies only to the array, and not to the linked node structure, which has a more liberal type. For instance, it is possible to reach T using the *lastChild* of ul.



Thus, we have shown how the language of this paper can be used to encode the core of the DOM, with the basic operations provided by DOM nodes, with high precision. In contrast to earlier work (e.g., [37]) this is achieved entirely without taking any DOM-specific information into account. Everything, including the arrays, has been implemented in the language, and obey the basic type rules of the system. We are, in a natural way, able to freely traverse the resulting hierarchy and access nodes via the *childNodes* array.

B. User tracking

Consider a login system combining the use of a shared secret and the use of a username and password. Once the latter has been presented, a nine-digit keypad and a challenge are presented. To authenticate, the user finds the correct response to the challenge using the shared secret, and uses the keypad, to enter the response. Such a system is currently employed by barclays.es using a response size of 2 decimal digits chosen from a shared secret of 100 such numbers.

At the same time the page uses Google Analytics to track usage.

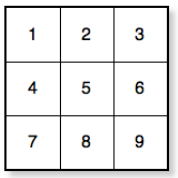
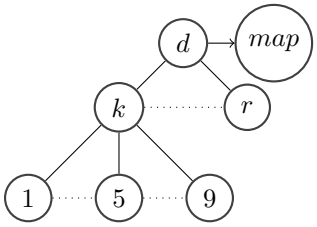


Figure 2. Keypad Example

Now assume, that the above is part of a bigger website, where some technology for generating click heat maps is applied. In such a case, it is very important that the position information of the clicks on the keypad is not propagated to the heap map generator, since the positions of the clicks give away parts of the shared secret. Further, one might want to protect not only the position information in the click events, but also the fact that the clicks occur. The reasons for this might include secrecy of the length of the response, or secrecy of the time at which the events occur.

We support the example by implementing a program modeling rudimentary support for event propagation. Without loss of generality, the model only supports one event listener per node, and one type of events. An extension to multiple event listeners and events is immediate, but clutters the model. Each node provides methods for setting the event listener, setting the parent and firing an event. The fire event method takes a parameter, the event data, and calls the registered event listener, if one exists. If the event listener returns *true* the event is propagated to the parent otherwise not. Three different fire event methods are provided: one acting as previously described, one upgrading the event data to secret, and one running the event handler under secret control. The second fire event method models events where the data is secret, e.g., the position data of the clicks discussed above, and the third models when both the data and the existence of the clicks is secret. In the following, consider the events to be the *onClick* events generated when the user click on the buttons of the keypad. Simplified, the above example can be represented with the following DOM node hierarchy, where the nodes 1 to 9 represents the buttons grouped under a keypad node k, which is part of the document d, together with the rest of the document r, and the click heat map generator map.



The hierarchy also represents the parent hierarchy of event propagation, i.e., events on 1 are propagated to k, and finally to d, where the event is collected by the click heat map generator, if the handlers do not stop propagation. Now consider the following different possibilities.

Unconstrained propagation: corresponds to the situation described above, where an event is injected using the first fire event method and is propagated upwards in the parent hierarchy after execution of the local handler until reaching the top. In the example this would allow deduction of the shared secret from the event data sent to the heat map generator.

Propagation of secrets: corresponds to the situation, where the second fire event method is used to inject the event. This handler upgrades the security label of the data of the event, i.e., the position of the click. This makes the event data secret. Even though the event is propagated upwards, any operation on the event data will reflect its secrecy, and if data reaches the heat map generator it will not be able to use the data without triggering a security violation. Hence, the map generator gets to know the existence of the event, but not its contents.

Stopped propagation of secrets: If we want to prohibit the heat map generator from receiving click events from the keypad altogether, we have two options. The first option is to install a non-propagating handler in k, (or all of 1 to 9) that stops the propagation of events upwards to the parent. In such case, events originating from 1 to 9 will be propagated to k, where the handler will return *false* preventing further propagation which stops the event from reaching the heap map generator.

Secret propagation: The other method allows the prevention of click events originating from 1 to 9 from being used by the heat map generation without installing special handlers in k. Instead, if the third fire event method is used, the event handlers, including the heap map generator, will be executed under secret control. The result is that the heat map generator is prevented from causing public side effects, and thus neither the event data or the fact that the event occurred can be used without causing a security violation. This method is preferable, since it relies on the security labels, rather than the presence of a special handler.

We have shown how a rudimentary event model can be encoded natively in the language, and how the features provide the possibility of dynamic event propagation hierarchies, in addition to both protecting the data of events, and the fact that the event occurred. The event implementation can be seen as a simplified version of the event model of Rafnsson and Sabelfeld [34], which extends the reactive model presented by Bohannon et al. [8].

VI. Related work

Amongst a large body of research on language-based approach to information-flow security [39], we discuss most related work on dynamic information control, as well as related work that targets securing JavaScript.

Dynamic information-flow control Our paper pushes the limits of dynamic information-flow enforcement on both expressiveness of the underlying language and on the permissiveness of the enforcement. We briefly discuss previous work that serves as our starting point.

Russo and Sabelfeld [36] show that purely dynamic flow-sensitive monitors do not subsume the permissiveness of flow-sensitive security type systems. Although our monitor is purely dynamic, the language includes a security label upgrade operator. This means that we can mimic type systems by injecting security upgrades in appropriate parts of the code. Hence, the permissiveness of our approach can be boosted to that of hybrid monitors, at the cost of programmer annotations.

Chugh et al. [10] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage.

Fenton [21] discusses purely dynamic monitoring for information flow but does not prove noninterference. Volpano [43] considers a purely dynamic monitor to prevent explicit (but not implicit) flows. In a flow-insensitive setting, Sabelfeld and Russo [40] show that a monitor similar to Fenton's enforces termination-insensitive noninterference without losing in precision to classical static information-flow checkers. This line of work has progressed further to extend the monitor to a language with dynamic code evaluation, communication, and declassification [1], as well as timeout instructions [35].

In previous work, Russo et al. [37] investigate the impact of dynamic tree structures like the DOM on information flow. The monitor focuses on preventing attacks based on navigating and deleting DOM tree nodes. The monitor derives the security level of existence for each node from the context of its creation. Our model can be viewed as a generalization, where the DOM falls out naturally, and without losing permissiveness, from the general treatment of pointers and linked structures.

Austin and Flanagan [2], [3] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Austin and Flanagan [3] discuss inserting *privatization operations*, which are akin to our upgrade commands. The insertion takes place when a variable that was previously upgraded in secret context is about to be branched upon. Magazinius et al. [30] show how to inline a no-sensitive upgrade monitor into programs in a language with dynamic code evaluation.

Bohannon et al. [8] present a flow-insensitive static analysis for JavaScript-like event systems. Rafnsson and Sabelfeld [34] model event hierarchies and present a hybrid of flow-sensitive static analysis and transformation that guarantees that at most one bit is leaked per consumed public input. Besides the natural differences on dynamic vs. static analysis, our event implementation can be seen as a simplified version of Rafnsson and Sabelfeld's event model.

JavaScript semantics The literature includes two major approaches to formalizing the semantics of JavaScript.

On one hand, Maffeis et al. [29] give a detailed semantics for full JavaScript. It is a full account of the ECMA-262 standard (v.3) [17], which faithfully models the, sometimes slightly unusual, behavior of JavaScript programs.

On the other hand, Guha et al. [25] present a semantics claimed to capture the essence of JavaScript. They provide a core functional language that shares some similarities to the semantics of Maffeis, but deviates in a number of important places regarding the modeling of variables and functions.

Yu et al. [45] also formulate a semantics in terms of a lambda calculus. Contrary to Guha et al. [25], no attempt at faithfully mimicking JavaScript scoping is made, thus avoiding key problems associated with JavaScript.

Our semantics is closest to that by Maffeis et al, with the obvious difference of instrumentation with information-flow checks. Nevertheless, we expect that our transparency theorem also holds against the semantics by Maffeis et al. Compared to the semantics of Guha et al., using a variable environment chain requires more heavyweight formalism. However, it has the benefit that it is able to deal with the entire (complex) scoping behavior of JavaScript, including *with*. This is challenging to model in the semantics of Guha et al., as noted by them. In addition, being close to the standard makes it natural to verify that the semantics is faithful to the JavaScript semantics.

Our subset of JavaScript is distilled to illustrate the main challenges for tracking information flow. Hence, our work is not a safe sub-language approach like Caja [31], FaceBook JavaScript (FBJS) [20], ADSafe [13], Gatekeeper [23], or the approach by Guha et al. [24]. Further, we chose to be faithful to the latest version (5) of the ECMA-262 standard.

Empirical JavaScript studies On the other side of the spectrum there is empirical work where the goal is not soundness but catching information-flow attacks in the wild. Vogt et al. [42] implement a static flow-sensitive information-flow analysis to crawl around 1,000,000 popular web sites and, after white/black-listing 30 web sites, detect suspected attempts for cross-domain communication in 1,35% of the sites.

Jang et al. [28] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. The analysis is based on code rewriting that inlines checks for data produced from sensitive sources no to flow into public sinks. They detect a number of attacks present in popular web sites, both in custom code and in third-party libraries.

We believe that the road to bridging the gap between formal and empirical approaches is dynamic information-flow tracking. For a language like JavaScript, static analysis is hardly feasible, as argued by this paper and others [41], while dynamic analyses provide possibilities for precisely

recording relations between data in a given trace. The FlowSafe [19] project at Mozilla takes a similar (dynamic) path.

Secure multi-execution In an orthogonal yet promising effort, Devriese and Piessens [16] investigate enforcement of secure information by multi-execution. Multi-execution runs the original program at different security levels and carefully synchronizes communication among them. Multi-execution is secure by design since programs that compute public input only get access to public input. Bielova et al. [6] implement secure multi-execution for the Featherweight Firefox [9] model. Austin and Flanagan [4] propose faceted values to model secure multi-execution within a single run. Each value facet corresponds to the view of the value from the point of an observer at a given security level. They show that this approach is semantically equivalent to secure multi-execution for a λ-calculus with mutable reference cells. An advantage of this approach with respect to multi-execution is that a single faceted execution simulates as many non-faceted executions as there are elements in the security lattice. However, faceted values have to deal with the problem of tracking control flow (which is a non-issue in original secure multi-execution). It is not clear how to scale faceted values to handle exceptions. For the secure multi-execution approach as a whole, it remains to be investigated whether silent modification of behavior with respect to the original program (such as reordering communication events) is an obstacle in practice.

VII. Conclusion

We have developed a dynamic type system for enforcing secure information flow for core features of JavaScript: objects, higher-order functions, exceptions, and dynamic code evaluation. Our semantic model closely follows the choices of the ECMA-262 standard (v.5) on the language constructs from the core. We have established a formal guarantee that the type system guarantees noninterference. In addition, we have illustrated our results with two prototype implementations that show that our framework is powerful enough to support functionalities of JavaScript's API from the document object model (DOM) without loss of precision. As a result, we improve on previous work both with respect to modeled language features and permissiveness of the enforcement.

As argued in Section II, our results provide a solid platform for tackling the rest of JavaScript. We believe that the technical material developed in this paper suffices to support the remaining constructs of JavaScript with minor extensions.

In accord with the formalization development, we pursue implementation in a form of a JavaScript execution monitor. Future work includes an evaluation of the permissiveness as well as the performance overhead of the monitor.

References

[1] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[2] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[3] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

[4] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2012.

[5] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[6] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. International Conference on Network and System Security (NSS)*, pages 97–104, Sept. 2011.

[7] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. Draft, Apr. 2012.

[8] A. Bohannon, B. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, Nov. 2009.

[9] A. Bohannon and B. C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Proc. USENIX Security Symposium*, June 2010.

[10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[11] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[12] Coq. http://coq.inria.fr.

[13] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.

[14] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[15] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[16] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.

[17] ECMA International. ECMAScript Language Specification, 1999. Version 3.

[18] ECMA International. ECMAScript Language Specification, 2009. Version 5.

[19] B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.

[20] Facebook. FBJS. http://wiki.developers.facebook.com/index.php/FBJS, 2009.

[21] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[22] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[23] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, Berkeley, CA, USA, 2009. USENIX Association.

[24] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, Washington, DC, USA, 2011. IEEE Computer Society.

[25] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, June 2010.

[26] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. Full version, http://www.cse.chalmers.se/~utter/jsflow/jsflow-full.pdf.

[27] A. L. Hors and P. L. Hegaret. Document Object Model Level 3 Core Specification. Technical report, The World Wide Web Consortium, 2004.

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

[29] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS'08*, volume 5356 of *LNCS*, pages 307–325, 2008.

[30] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proceedings of the IFIP International Information Security Conference (SEC)*, Sept. 2010.

[31] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.

[32] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[33] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.

[34] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2011.

[35] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[36] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[37] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.

[38] P. D. Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joose. Security of web mashups: a survey. In *Nordic Conference in Secure IT Systems*, LNCS, 2010.

[39] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[40] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[41] A. Taly, U. Erlingsson, M. Miller, J. Mitchell, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE Symp. on Security and Privacy*, May 2011.

[42] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.

[43] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[44] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[45] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 237–249. ACM, 2007.