Journal of Computer Security 0 (2016) 1–54
IOS Press

Information-flow security for JavaScript and its APIs

Daniel Hedin [a,b], Luciano Bello [a] and Andrei Sabelfeld [a,*]
[a] *Chalmers University of Technology, Department of Computer Science and Engineering*
[b] *Mälardalen University, School of Innovation, Design and Engineering*

Abstract. JavaScript drives the evolution of the web into a powerful application platform. Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services built up from third-party code. Script inclusion poses a challenge of ensuring that the integrated third-party code respects security and privacy.

This paper presents a dynamic mechanism for securing script executions by tracking information flow in JavaScript and its APIs. On the formal side, the paper identifies language constructs that constitute a core of JavaScript: dynamic objects, higher-order functions, exceptions, and dynamic code evaluation. It develops a dynamic type system that guarantees information-flow security for this language. Based on this formal model, the paper presents *JSFlow*, a practical security-enhanced interpreter for fine-grained tracking of information flow in full JavaScript and its APIs. Our experiments with JSFlow deployed as a browser extension provide in-depth understanding of information manipulation by third-party scripts. We find that different sites intended to provide similar services effectuate rather different security policies for the user's sensitive information: some ensure it does not leave the browser, others share it with the originating server, while yet others freely propagate it to third parties.

Keywords: web application security, JavaScript, information flow, reference monitoring, noninterference

1. Introduction

Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services, often utilizing third-party code. Such code provides a range of facilities from utility libraries (such as jQuery) to readily available services (such as Google Analytics and Tynt). Even stand-alone services such as Google Docs, Microsoft Office 365, and DropBox offer integration into other services. Thus, the web is gradually being transformed into an application platform for integration of services from different providers.

Motivation: Securing JavaScript At the heart of this lies JavaScript. When a user visits a web page, even a simple one like a loan calculator or a newspaper website, JavaScript code from different sources is downloaded into the user's browser and run with the same privileges as if the code came from the web page itself. This opens up possibilities for abuse of trust, either by direct attacks from the included scripts

*Corresponding author

0926-227X/16/$27.50 © 2016 – IOS Press and the authors. All rights reserved

or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. A large-scale empirical study [55] of script inclusion reports high reliance on third-party scripts. As an example, it shows how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains. A representative real-life example is an attack on Reuters in June 2014, attributed to "Syrian Electronic Army", which compromised a third-party widget for content-targeting (Taboola) to redirect visitors from Reuters to another web site [66]. This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

At the same time, the business model of many online service providers is to give away a service for free while gathering information about their users and their behavior in order to, e.g., sell user profiles or provide targeted ads. How do we draw a line between legitimate information gathering and unsolicited user tracking?

This poses a challenge: how to enable exciting possibilities for third-party code integration while, at the same time, guaranteeing that the integrated third-party code respects the security and privacy of web applications?

Background: State of the art in securing JavaScript Today's browsers enforce the *same-origin policy (SOP)* in order to limit 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. SOP prevents direct communication with non-origin domains but allows indirect communication. For example, sensitive information from the user can be sent to a third-party domain as part of an image request.

Although loading a script in an iframe provides secure isolation, it severely limits the integration of the loaded code with the main application. Thus, the iframe-based solution is a good fit for isolated services such as context-independent ads, but it is not adequate for context-sensitive ads, statistics, utility libraries, and other services that require tighter integration.

Loading a script via a script tag provides full privileges for the included script and, hence, opens up for possible attacks. The state of the art in research on JavaScript-based secure composition [62] consists of a number of approaches ranging from isolation of components to their full integration. Clearly, as much isolation as possible for any given application is a sound rationale in line with the principle of least privilege [65]. However, there are scenarios where isolation incapacitates the functionality of the application.

As an example, consider a loan calculator website. The calculator requires sensitive input from the user, such as the monthly income or the total loan amount. Clearly, the application must have access to the sensitive input for proper functionality. At the same time, the application must be constrained in what it can do with the sensitive information. If the user has a business relationship with the service provider, it might be reasonable for this information to be sent back to the provider but remain confidential otherwise. However, if this is not the case, it is more reasonable that sensitive information does not leave the browser. How do we ensure that these kinds of fine-grained policies are enforced?

As another example, consider a third-party service on a newspaper site. The service appends the link to the original article whenever the user copies a piece of text from it. The application must have access to the selected text for proper functionality as the data must be read, modified and written back to the clipboard. However, with the current state of the art in web security, any third-party code can always send information to its own originating server, e.g., for tracking. When this is undesired, how do we guarantee that this trust is not abused to leak sensitive information to the third party?

Unfortunately, access control is not sufficient to guarantee information security in these examples. Both the loan calculator and newspaper code must be given the sensitive information in order to provide the intended service. The *usage* of sensitive information needs to be tracked *after* access to it has been granted.

Goal: Securing information flow in the browser These scenarios motivate the need of information-flow control to track how information is used by such services. Intuitively, an information-flow analysis tracks how information flows through the program under execution. By identifying sources of sensitive information, an information-flow analysis can limit what the service may do with information from these sources, e.g., ensuring that the information does not leave the browser, or is not sent to a third party.

The importance of tracking information flow in the browser has been pointed out previously, e.g., [46,73]. Further, several empirical studies [70,39,32,30] (discussed in detail in Section 9) provide clear evidence that privacy and security attacks in JavaScript code are a real threat. The focus of these studies is *breadth*: trying to analyze thousands of pages against simple policies.

Complementary to the previous work, our overarching goal is an in-*depth* approach to information-flow tracking in practical JavaScript. Accordingly, we separate the goal into fundamental and practical parts:

First, our goal is to resolve the fundamental challenges for tracking information flow in a highly dynamic language at the core of JavaScript with features such as dynamic objects, higher-order functions and dynamic code evaluation. This goal includes a formal model of the enforcement mechanism in order to guarantee secure information flow.

Second, our goal is to push the fundamental approach to practice by extending the enforcement to full JavaScript, implementing it, and evaluating it on web pages where in-depth understanding of information-flow policies is required.

Objectives: JavaScript and libraries The dynamism of JavaScript puts severe limitations on static analysis methods [67]. These limitations are of particular concern when it is desirable to pinpoint the source of insecurity. While static analysis can be reasonable for empirical studies with simple security policies, the situation is different for more complex policies. Because dynamic tracking has access to precise information about the program state in a given run, it is more appropriate for in-depth understanding of information flow in JavaScript. With this in mind, our overarching goal translates into the following concrete objectives.

1. The first objective is to identify a core of JavaScript that embodies the essential constructs from the point of view of information-flow control. We set out to propose dynamic information-flow control for the code, with the technical challenges expanded in Section 3.
2. The second objective is covering the *full non-strict JavaScript* language, as described by the ECMA-262 (v.5) standard [26]. This part draws on the sound analysis for the core of JavaScript, and so the challenge is whether the rich security label mechanism and key concepts such as read and write contexts scale to the full language.
3. The third objective is covering *libraries*, both JavaScript's built-in objects, and the ones provided by browser APIs. The *Document Object Model (DOM)* API, a standard interface for JavaScript to interact with the browser, is particularly complex. This challenge is substantial due to the stateful nature of the DOM. Attempts to provide "security signatures" to the API result in missing security-critical side effects in the DOM. The challenge lies in designing a more comprehensive approach.
4. The fourth objective is implementing the JavaScript interpreter in JavaScript. This allows the interpreter to be deployed as a Firefox extension by leveraging the ideas of Zaphod [53]. The interpreter

keeps track of the security labels and, whenever possible, it reuses the native JavaScript engine and standard libraries for the actual functionality.

5. The fifth objective is evaluation of the approach on scenarios where in-depth understanding of modern third-party scripts (such as Google Analytics) is required. Following the scenarios mentioned above, the focus of the evaluation is on loan calculator web sites and web sites with behavioral tracking.

Contributions and overview After recalling the basics of dynamic information-flow control (Section 2), we spell out the challenges of securing JavaScript (Section 3). The challenges are divided into three sections: challenges related to the language, challenges related to a full-scale implementation, and challenges related to dynamic enforcement. Together, the challenges justify the choice of the JavaScript core, and illustrate key issues related to a practical implementation.

Addressing the first objective, we identify a language that constitutes a core of JavaScript. This language includes the following constructs: dynamic objects, higher-order functions, exceptions, and dynamic code evaluation (Section 4). We argue that the choice of the core captures the essence of the language. It allows us to concentrate on the fundamental challenges for securing information flow in dynamic languages. 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) [26] on the language constructs from the core.

We develop a dynamic type system for information-flow security for the core language (Section 4) and establish that the type system enforces a security policy of *noninterference* [19,29], that prohibits information leaks from the program's secret sources to the program's public sinks (Section 5).

Addressing the second objective, we scale up the enforcement to cover the full JavaScript language (Section 6). The scaled-up enforcement tightly follows the enforcement principles for the core, indicating that the choice of the core is well-justified, and implements the solutions outlined by the implementation challenges.

Addressing the second and fourth objectives, we report on the implementation of the *JSFlow*, an information-flow interpreter for full non-strict ECMA-262 (v.5) (Section 6). JSFlow is itself implemented in JavaScript. This enables the use of JSFlow as a Firefox extension, *Snowfox*, as well as on the server side, e.g., by running on top of *node.js* [40]. The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [52] passed by SpiderMonkey and V8.

Addressing the third and fourth objectives, we have designed and implemented extensive stateful information-flow models for the standard API (Section 7). This includes all of the standard API, as well as the API present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. A distinction is made between *shallow* and *deep* models, which represent different trade-offs between reimplementing native code and maintaining a model of its information flow.

To the best of our knowledge, this is the first effort that bridges all the way from theory to implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

Addressing the fifth objective, we report on our experience with JSFlow/Snowfox for in-depth understanding of existing flows in web pages (Section 8). Rather than experimenting with a large number of web sites for simple policies (as done in previous work [70,39,32,30]), we focus on in-depth analysis of two case studies.

The case studies show that different sites intended to provide similar service (a loan calculator) enforce rather different security policies for possibly sensitive user input. Some ensure it does not leave the browser, others share it only with the originating server, while yet others freely share it with third party

services. The empirical knowledge gained from running the case studies on actual web pages and libraries has been key for understanding the possibilities and limitations of dynamic information-flow tracking and setting future research directions.

The paper merges, expands, and improves our previously disjoint efforts on theory [37] and implementation [36] of information-flow control for JavaScript. In addition to unifying these efforts, the following are the most prominent features that offer value over the previous work.

- We have reworked the rules for *Delete* and *Put* of the ECMA objects to make them more intuitive and to make their relation clearer.
- We have added the full proof of soundness of the core language.
- We have significantly reworked several sections to improve the readability of the paper. In particular, we have 1) added Section 2 on dynamic information flow, 2) reworked Section 3 to more clearly state the challenges and their solutions, as well as added some examples and challenges, and 3) performed a significant expansion of Section 4 on the core language.
- Based on our practical experiments we have reworked the upgrade instructions. Upgrading to a predetermined static label is sometimes inflexible in practice. It is frequently not possible determine which label to upgrade to. Instead, we employ more flexible upgrade instructions that allow the upgrade to be based on the dynamic label of a value.

2. Dynamic information flow

Before presenting the security challenges, we briefly recap standard information-flow terminology [22]. 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*. In the following examples, h and l represent high and low variables. This two-level model can be generalized to arbitrary security lattices [21].

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. In order to handle implicit flows, a security level associated with the control flow is introduced, called the *program counter level*, or *pc* for short. In the above example, 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.

The *pc* reflects the confidentiality of guard expressions controlling branch points in the program, and governs side effects in their branches by preventing modification of less confidential values.

Dynamic information-flow enforcement Dynamic information-flow analysis is similar to dynamic type analysis. *Security labels* store security levels for associated variables. Each value is labeled with a security label representing the confidentiality of the value. Dynamic information-flow enforcement is naturally *flow sensitive* because the security label is associated with runtime values. Consider the following program:

```
l = h;
```

Although the original value of l is labeled public, the assignment will overwrite both the value and the security label with the value and security label of h. This discipline makes dynamic tracking highly suitable for explicit flows.

However, flow sensitivity of dynamic analyses has a known limitation [28,60] related to implicit flows. Consider the following example, where we assume h is either 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.

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* (NSU) [3] discipline. For soundness, security violations force execution to halt. Hence, the execution of the above program results in a security error.

Read and write contexts The notions of *read context* and *write context* facilitate the explanation of dynamic information-flow enforcement in a language with references. These notions govern all information-flow control rules in this paper.

The read context for an entity is the accumulated security label of the *access path* of the entity. For a property, the access path is the primitive reference to the object containing the property together with property name. When reading, the result is raised by security label of the read context.

To illustrate the notion of access path and read context, consider the following program. The access path to the property identified by the value of p in o is the security label of o together with the security label of p.

```
x = o[p];
```

The security label of the value written to x is the security label of the value stored at the read property raised by the security label of o and p.

The write context of an entity is the read context together with the *security context*, i.e., the program counter. When writing to an entity, NSU demands that the security label of the entity is at least as secret as its write context.

The notion of write context is illustrated in the following program, where the access path is the same as in the example above, but where the write additionally takes place under the control of x.

```
if (x) {
  o[p] = y;
}
```

In this example, the demand from the write context translates into the demand that the label of the target is above the labels of x, o and p.

3. Challenges

The ultimate goal of this work is practical enforcement of information-flow security in JavaScript. This entails both a solid theoretical base, as well as a full scale implementation. Reflecting this, this section is divided into three parts. First, we outline the main language challenges of JavaScript. This illustrates the core language presented in Section 4. Second, we outline the challenges of extending the core language to full JavaScript as defined in the ECMA-262 (v.5) [26] standard (sometimes referred to as *the standard* henceforth) and how they relate to the core challenges. Finally, we discuss challenges relating to dynamic enforcement of secure information flow. For each of the challenges we illustrate the problem and give our approach to solving it.

3.1. 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 (v.5) [26] 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 dynamic objects, prototype inheritance, higher-order functions, exceptions, and dynamic code evaluation. 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.

3.1.1. Dynamic code evaluation: 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 analysis, since the string may not be available to a static analysis.

Our approach: Dynamic analysis does not share this limitation, by virtue of the fact that dynamic analysis executes at runtime.

3.1.2. Aliasing and flow sensitivity

The type system of JavaScript is *flow sensitive*: the types of, e.g., variables and properties 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 analysis, 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 property f of the newly allocated object is set to 0, and y is made an alias of x. Finally, the property f is overwritten by a secret via y. This kind of programs are rejected by static analysis like Jif [54]. In the dynamic setting, we simply update the security label of the value of the property f of the allocated object.

3.1.3. Dynamic objects: structure and existence

JavaScript objects are dynamic in the sense that properties can be added to or removed at runtime. This implies that not only the values of property can be used to encode information, but also their presence or absence. Consider the following example:

```
o = {};
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 both the presence and 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, e.g., `l = (o.q != undefined);` (projecting non-existing properties returns undefined).

Our approach: We solve this by associating an *existence security label* with every property and a *structure security label* with every object. The demand is that changing the structure of the object by adding or removing properties under secret control is possible only if the structure security label of the object is secret. Once the structure of an object is secret, knowing the absence of properties in the object is also secret.

3.1.4. 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 second update is run or not and, hence, l encodes information about h.

Our approach: We introduce a security label for exceptions, the *exception security label*. The exception security label is an upper bound on the label of the security context in which exceptions are allowed. If the exception security label is public, exceptions under secret control are prohibited. The exception security label and the *pc* together form the security context that governs side effects. That is, if the exception security label is secret, the security context is secret, and side effects are constrained.

3.1.5. 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.

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

Depending on the secret h, a function that writes 1 or a function that writes 0 to l is chosen. Calling the selected function 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 [56].

3.1.6. 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 assigning to undeclared variables causes the variable to be defined globally, cause complex and sometimes unexpected behavior. To appreciate this, consider, e.g, 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 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 conditional 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.

A similar situation can be created using *with* that takes an object and injects it into the variable chain allowing the properties of the injected object to shadow variables declared higher up in the chain. Consider the following example:

```
o = {};
if (h) { o['l'] = 1 }
with (o) {
  l = 0;
}
```

When the update `l = 0;` is executed, the variable lookup will first look in the topmost environment record, i.e., the injected o object. If the object defines a property with the name l, the variable lookup terminates returning a reference to the object. Otherwise, the lookup continues in the remaining variable environment chain. In the above example, in case h is true the update is captured by o. If h is false, l is created in the global variable environment.

The result is that both the value of l in the global variable environment and potentially its existence encode information about h. The latter implies that, similar to objects, we must track the structure of variable environments. In addition, the presence or absence of the l in the local variable environment (or elsewhere in the chain) effects where the update takes place.

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 basic objects and object operations.

3.1.7. Prototype hierarchy

In a similar way, the prototype chain, forming the basis for prototype based inheritance, can cause intricate behavior. In JavaScript, functions are used to create new objects. The body of the function acts as a constructor for the newly created object, and the contents of the *prototype* property is copied into the prototype chain of the object. The prototype chain is then traversed when looking up a property in the object. This is done in a similar manner to the variable chain. Consider the following example:

```
function f() { };
f.prototype.l = 0;

var x = new f();
```

Executing `x['l']` would return 0 even though x does not define the property l itself. Instead, it is the property l of the prototype that is returned. Consider the following addition to the above example:

```
function g() { };
g.prototype = x;

x = new g();
```

We create a new function g and let the prototype field of g be x, i.e., the object created by f, and we let x be an object created by g. Executing `x['l']` would result in 0 even though neither x nor the immediate prototype of x defines l. The prototype of the prototype of x does, however, and its property l is found and its value, 0, is returned. In this way, x inherits the properties of all prototypes in the prototype chain. Now, consider the following addition to the example:

```
if (h) {
  g.prototype.l = 1;
}
```

Since the prototype chain is traversed for each property lookup, modifications of the prototypes in the prototype chain may cause effects on the property lookup. In case h is true, the property l is added to the immediate prototype of x. This shadows the previous property l, and executing `x['l']` now returns 1 instead of 0.

Our approach: We show that the intricacies of prototype inheritance can be handled by a faithful modeling in terms of basic objects and object operations.

3.2. Tackling full JavaScript

A practical implementation of secure information-flow enforcement for JavaScript requires that the core solutions are extended to the full language and execution environment as defined by the standard. We outline the most interesting extensions relating to the handling of the full standard.

3.2.1. Return labels

The core language makes the simplifying assumption that there is a unique *return* statement at the end of each function. When this assumption is not met in actual code, there are similar challenges to the ones created by exceptions. Consider the following program:

```
l = true;
function f() {
  if (h) { return 1; }
  l = false;
}
```

The program copies the secret boolean h into l by returning from the function under secret control. If h is true, the assignment `l = false` is not executed, and l remains true. If h is false the assignment is performed, and l becomes false. Thus, the *return* statement has the effect of extending the secret context to the end of the function.

Our solution: we introduce a *return label* that governs *return* statements. This label is similar to the exception label in that it defines an upper bound on the control contexts in which `return` can be executed. The return label is also part of the security context. Hence, a secret return label causes the security context to be secret which constrains side effects.

3.2.2. Labeled statements

JavaScript contains labeled statements and allows jumping to them using `break` and `continue`. As with conditional statements, such transfer of control may result in implicit flows. Consider the following example:

```
l=true;
L1: do {
  if (h) { continue L1; }
  l=false;
} while(false);
```

If h is true, the *continue* statement causes *do-while* to proceed to evaluating the guarding expression, which, in this case, causes the termination of the loop. This prevents the execution of the assignment `l=false`, and l remains true. If h is false, the assignment is performed, resulting in l becoming false.

Our solution: we associate each statement label with a security label. This label is similar to the exception label and return label in that it defines an upper bound on the control contexts in which jumping to the associated statement is allowed. The security label of labeled statement is also a part of the control context for the labeled statement itself, which constrains side effects.

3.2.3. Accessor properties

In addition to standard value properties, JavaScript supports accessor properties. Accessor properties allow overriding property reads and writes with user functions. When reading, the return value of the *getter* function of the property is returned, and, similarly, writing to a property invokes the *setter* function associated with the property. Section 7 shows how getters and setters can be used in complicated interplay with libraries, opening the door for non-obvious information flows.

Our solution: handling accessor properties is akin to handling first-class functions. Instead of storing the actual value in the property, the accessor functions are stored.

3.2.4. Implicit coercions

Many expressions, statements, and API functions of JavaScript convert their arguments to primitive types. JavaScript objects may override this conversion process with user functions. Consider the following example:

```
l = false;
x = { valueOf : function ()
              { return h ? {} : 1; },
      toString : function()
              { l = true; return 1; } };
h = x + 1;
```

The addition operation tries to convert its arguments to primitive values. For an object, the conversion first invokes its *valueOf* method, if present. If this returns a primitive value, it is used. If not, the *toString* method is invoked instead. In the example, *valueOf* is chosen so that *toString* is invoked only when h is true, which must therefore be reflected in the control context of *toString*.

Our solution: the information flow introduced by implicit coercions originate from *interpreter-internal* information flow, i.e., when the interpreter itself inspects labeled values and choses whether or not to invoke the coercion function. Interpreter-internal flows are not limited to the implementation of implicit coercions. They are handled uniformly by generalizing the notion of the *pc*, see Section 6.

3.3. Libraries

Handling the full JavaScript standard implies handling the standard execution environment. This can be challenging due to the interplay between the computation performed by the library and different language features.

State Libraries may contain internal state that must be modeled. The prime example of this is Document Object Model (DOM) API provided by modern browsers, where the rendered document is part of the internal state. Much of the standard API is also stateful, including *Object*, *Function*, *Array*, and *RegExpr*.

Callbacks Libraries frequently make use of callbacks. One source of this is implicit coercions performed on the arguments. Other sources include registered callbacks (e.g., the event handlers of the DOM) or indirect callbacks via getters and setters on objects passed in as arguments. Of these, boundary conversion is relatively easy to handle efficiently, and for registered callbacks it is often possible to create precise models of the security context and argument security labels of the callback. Due to the intricate interaction with the internal information flow of the library handling, getter and setters in the library setting are frequently challenging.

Our solution: we introduce the concept of *shallow* and *deep* models to aid in the analysis of library models. In practice, each library is fitted with a tailor-made dynamic information-flow model, see Section 7.

3.4. Dynamic flow sensitivity

As discussed in Section 2, pure dynamic enforcement of secure information flow puts limitations on how security labels are allowed to change under secret control. This paper adheres to the NSU paradigm and prohibits security labels from changing under secret control. This introduces a potential source of inaccuracies in the enforcement, since programs like the following will be stopped with a security error.

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

From a practical perspective this is unfortunate, since it may cause secure programs to be stopped with a security error.

Our solution: we solve this problem by extending the language with upgrade instructions. The upgrade instruction are dynamic in the sense that they allow labels to be dynamically upgraded to match the label of other value. Consider for instance the upgrade expression $upg\ e_1\ to\ e_2$, that upgrades the label of the value that e_1 evaluates to the label of e_2. Consider its use in the above example:

```
l = upg false to h;
if (h) {
  l = true;
}
```

After the upgrade the value stored in l, it will have the same label as h, which guarantees that the execution is not stopped.

In addition to the upgrade expression, we provide upgrade statements for upgrading the structure security label of object, the existence security label of properties, the structure security label of environment records, and the exception security label. The implementation provides additional upgrade statements for upgrading the return security label and the security labels associated with statement labels.

4. Theory of information-flow security for JavaScript

The language 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. All the presented results can be generalized to arbitrary security lattices, but this requires that all definitions are parameterized by the attacker level. The intuition for the generalization is that everything at or below the level of the attacker is to be treated as public, and everything else is to be treated as secret. The generalization is mostly mechanical but clutters the definitions and proofs significantly, while providing little additional insight. The implementation uses a general powerset lattice of origins, see Section 6.

Potentially insecure flows are prevented by stopping the execution. This 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 per execution, which is reflected in our baseline security condition in Section 5.

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

Fig. 1. Syntax

4.1. Syntax

The syntax of the language can be found in Figure 1. It consists of two basic syntactic categories: expressions and statements. Let $\overline{X}$ range over lists of X, where $\cdot$ denotes the cons operator and $[\,]$ denotes the empty list. As an example, $\overline{x}$ denotes lists of variable names, and $\overline{e}$ denotes lists of expressions.

Expressions, e, are built up by literals, variables, self reference in the form of this, property projection, assignment, delete, typeof, binary operators, function application, object construction, named function expressions, and an upgrade expression for values, upg. The literals consist of labels, l, strings, s, numbers, n, and booleans, b, together with the distinguished *undefined* and *null*.

As discussed above, without loss of generality, the labels are either L or H, i.e., $l ::= L \mid H$ where $L < H$.

For brevity, the unary operators are represented by the *delete* and *typeof* operators, and the binary operators are opaquely represented by $\star$. The *delete* operator provides a way of deleting variables and properties of objects, and the *typeof* operator returns a string representing the type of the argument. The omitted operators pose no significant information-flow challenges.

Statements, ranged over by c, are built up by variable declaration, sequencing, conditional choice, iteration via *while* and *for-in*, exception throwing and catching, the *return* statement for returning values from functions, as well as *eval* and *with* providing dynamic code evaluation and lexical environment extension, respectively. The empty statement is represented by a distinguished *skip* statement, and expressions are lifted to statements. In addition, there are four upgrade statements that allow for the upgrade of the existence security label of properties, the structure security label of objects, as well as the upgrade of the exception security label, and the structure label of variable environments.

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. Without risk of confusion, we identify string literals and variable names, writing s instead of "s".

4.2. Semantics

This section consists of two parts. First, we introduce 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, and end with a section on the semantics of the upgrade expressions and statements.

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.

4.2.1. Values

Let H (secret) and L (public) be security labels, ranged over by σ. The security labels are used to label the values with security information. For clarity in the semantic rules, let pc and ϵ range over security labels denoting the *program counter label* and the *exception security label*, respectively.

$$v ::= r \mid s \mid n \mid b \mid \mathit{undefined} \qquad p ::= \dot{v} \mid \mathcal{F} \mid c \mid \overline{x} \qquad o ::= \{s_1 \overset{\sigma_1}{\mapsto} p_1, \ldots, s_n \overset{\sigma_n}{\mapsto} p_n, \sigma\}$$
$$\phi : r \to o \qquad \mathcal{C} ::= (\sigma, \dot{r}_1, \dot{r}_2) \qquad E ::= (\phi, \sigma)$$

Fig. 2. Values

Values are defined in Figure 2. Primitive values, ranged over by v, are *primitive references*, strings, numbers, booleans, and the *undefined* value. Similarly to security labels, r ranges over primitive references in general, while le, ve, and τ range over primitive references denoting *lexical environments*, *variable environments*, and the *this binding*, defined below.

In the 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 $\dot{v}^{\sigma_2} = v^{\sigma_1^{\sigma_2}} = v^{\sigma_1 \sqcup \sigma_2}$.

Let the *property names* be strings ranged over by s. *References*, $(\dot{r}, \dot{s})$, are pairs of security labeled primitive references and strings, denoting the property named s in the object referred to by r. We let $\dot{v}$ range over both security-labeled values and references. Note that the references are not themselves labeled.

In addition to the primitive values, the notion of primitive object, ranged over by o, forms the foundation of the semantics. Let p range over *property values*. A primitive object is a collection of *properties*, where each property is an association between a property name and a property value. Each property is decorated with an *existence* security label and objects carry a *structure* security label. When seen as maps, the domain of an object consists of a set of strings instead of a set of security labeled strings. This reflects that existence security labels carry information about the existence of the property and not of the property name. The properties are either *internal* or *external* indicated by the $\mathit{IsExternal}$ predicate on property names, which is false for strings of the form $\_s\_$ and true otherwise. Internal properties are used in the implementation of the semantics but are not exposed to the programmer — see the semantics of *for-in* (ITER-2, and ITER-3) in Figure 7. The values of external properties are security-labeled primitive values, while internal properties may hold *algorithms* represented by general functions, $\mathcal{F}$, statements, c, and lists of formal parameters, $\overline{x}$.

Heaps, ranged over by ϕ, are mappings from primitive references to primitive objects. Pairs of heaps and exception security labels form the *execution environments*, ranged over by E. Let $E[r] = \phi[r]$ for $E = (\phi, \epsilon)$. In addition, execution takes place with respect to an execution *context* $\mathcal{C}$, 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. The lexical environment and the variable environment are built up as chains of environment records, see Section 4.2.3. In each function call, the lexical environment and the variable environment point to the same environment record, see *FunctionCall* in Section 4.2.7. The difference between the two is that the lexical environment is used to introduce local scope for binding of temporary variables, see rule TRY-2 in Section 4.2.10, while the variable environment always points to the initial record. Somewhat contrary to what is indicated by the names, the lexical environment is used for variable lookup, see rule VAR in Section 4.2.9, while the variable environment is used for variable hoisting in *eval*, see rule EVAL in Section 4.2.10. This has the effect that variables introduced by *eval* are hoisted to the toplevel in each function call.

4.2.2. Semantics of ECMA objects

The ECMA objects provide the foundation of the objects of the JavaScript execution environment. They are built on top of the primitive objects and provide a number of internal algorithm properties that provide a common interface for interacting with the object. In particular, the ECMA objects provide sup-

$$\text{GOP-1}\ \frac{o = E[r] \quad o = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots\}}{GetOwnProperty(r^{\sigma_1}, s^{\sigma_2})\ E = \{value \mapsto \dot{v}^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}\}}$$

$$\text{GOP-2}\ \frac{o = E[r] \quad s \notin dom(o) \quad o = \{\dots, \sigma_3\}}{GetOwnProperty(r^{\sigma_1}, s^{\sigma_2})\ E = undefined^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}}$$

$$\text{GP-1}\ \frac{GetOwnProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}}{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}} \qquad \text{GP-2}\ \frac{\begin{array}{c}GetOwnProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma_1} \\ o = E[r] \quad o[\_Prototype\_] = null^{\sigma_2}\end{array}}{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma_1 \sqcup \sigma_2}}$$

$$\text{GP-3}\ \frac{\begin{array}{c}GetOwnProperty(\dot{r}_1, \dot{s})\ E = undefined^{\sigma_1} \\ o = E[r_1] \quad o[\_Prototype\_] = \dot{r}_2 \quad GetProperty(\dot{r}_2, \dot{s})\ E = d\end{array}}{GetProperty(\dot{r}_1, \dot{s})\ E = d^{\sigma_1}}$$

$$\text{HP-1}\ \frac{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto p^{\sigma}\}}{HasProperty(\dot{r}, \dot{s})\ E = true^{\sigma}} \qquad \text{HP-2}\ \frac{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma}}{HasProperty(\dot{r}, \dot{s})\ E = false^{\sigma}}$$

$$\text{DEL-1}\ \frac{\begin{array}{c}o_1 = \phi[r] \quad o_1 = \{s \overset{\sigma_3}{\mapsto} \dot{v}, \dots, \sigma_4\} \quad o_2 = o_1 \setminus s \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \qquad \sigma <: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash Delete(r^{\sigma_1}, s^{\sigma_2})\ (\phi, \epsilon) = (true^{L}, (\phi[r \mapsto o_2], \epsilon))}$$

$$\text{DEL-2}\ \frac{o = E[r] \quad s \notin dom(o)}{pc \vdash Delete(\dot{r}, \dot{s})\ E = (true^{L}, E)}$$

$$\text{GET-1}\ \frac{GetProperty(\dot{r}, \dot{s})\ E = \{value \mapsto \dot{v}\}}{Get(\dot{r}, \dot{s})\ E = \dot{v}} \qquad \text{GET-2}\ \frac{GetProperty(\dot{r}, \dot{s})\ E = undefined^{\sigma}}{Get(\dot{r}, \dot{s})\ E = undefined^{\sigma}}$$

$$\text{PUT-1}\ \frac{\begin{array}{c}o_1 = \phi[r] \quad o_1 = \{s \overset{\sigma_3}{\mapsto} v_2^{\sigma_4}, \dots, \sigma_5\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ o_2 = o_1[s \overset{\sigma}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma <: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash Put(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

$$\text{PUT-2}\ \frac{\begin{array}{c}o_1 = \phi[r] \quad o_1 = \{s \overset{\sigma_3}{\mapsto} v_2^{\sigma_4}, \dots, \sigma_5\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ o_2 = o_1[s \overset{\sigma_3}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma \not<: \sigma_3 \qquad \sigma <: \sigma_4\end{array}}{pc \vdash Put(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

$$\text{PUT-3}\ \frac{\begin{array}{c}o_1 = \phi[r] \quad o_1 = \{\dots, \sigma_3\} \quad \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \\ s \notin dom(o_1) \qquad o_2 = o_1[s \overset{\sigma}{\mapsto} \dot{v}_1^{\sigma}] \qquad \sigma <: \sigma_3\end{array}}{pc \vdash Put(r^{\sigma_1}, s^{\sigma_2}, \dot{v}_1)\ (\phi, \epsilon) = (\phi[r \mapsto o_2], \epsilon)}$$

Fig. 3. ECMA objects

port for *prototype inheritance* exemplified in Section 3.1.7. The *prototype hierarchy* is built when a function is used to construct a new object, see *FunctionConstruct* in Section 4.2.7. In the process, the contents of the *prototype* property of the function object is copied to the internal *_Prototype_* property of the constructed object. This hierarchy is then traversed by the internal ECMA object algorithm *GetProperty* when looking for a given property.

Most of the standard is described in terms of this interface. Few algorithms manipulate 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.

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

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

The definitions of $GetOwnProperty$, $GetProperty$, $HasProperty$, $Delete$, Get, and Put are found in Figure 3. Let d range over property descriptors or *undefined*, $d ::= \{value \mapsto v\}|undefined$, and let d^σ be defined structurally as $\{value \mapsto v^\sigma\}$ or $undefined^\sigma$. Property descriptors are used in *GetOwnProperty* to distinguish between a defined property with value *undefined* and an undefined property.

GetOwnProperty Given a primitive reference and property name, *GetOwnProperty* returns a property descriptor, $\{value \mapsto v\}$, containing the value of the property (GOP-1) or $undefined$ in the case the property does not exists (GOP-2). In both cases, the security label of the result is raised to the read context, i.e., the primitive reference and property name. As discussed in Section 3.1.3, the structure security label is taken into account for non-existing properties.

GetProperty *GetProperty* is formulated in terms of *GetOwnProperty*. The former follows the prototype chain [26] while searching for the property. When looking for a property, the object is first consulted. If the object defines the property, the property descriptor returned by *GetOwnProperty* is returned (GP-1). Otherwise, the search continues in the prototype of the object, if present (GP-3). If the property is not found before the end of the prototype chain has been reached, *undefined* is returned (GP-2). During the search, the read context, i.e., the accumulation of the security labels of the primitive references and of the *GetOwnProperty* results, is computed and used to raise the final result.

HasProperty *HasProperty* is a boolean valued wrapper around *GetProperty*. It returns true, if the property exists, (HP-1) and false otherwise (HP-2).

Delete *Delete* deletes properties of objects. Given a primitive reference to a primitive object and a property name, the property is removed from the object. Deleting an existing property from an object (DEL-1) changes the structure of the object. Therefore, the write context of the object has to be below the structure label, and below the existence label of the deleted value. Deleting a nonexistent property does not change the object, and, thus, no demands are placed (DEL-2).

Get Given a primitive reference and a property name, *Get* uses *GetProperty* to obtain a property descriptor and returns the value of the property (GET-1) or *undefined*, if the property does not exist (GET-2).

Put Of the ECMA object algorithms, *Put* is the most complicated from an information-flow perspective. *Put* allows for the addition of new object properties and the update of existing object properties, with different information-flow restrictions. In the case of addition (PUT-3), 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 property, $pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2$, where σ_1 and σ_2 are the security labels of the primitive reference and the property name, respectively. The existence security label and the value security label are both raised to the write context.

In the case of update (PUT-1 and PUT-2), the structure of the object is not changed, and, hence, there is no demand on the structure security label of the object. With respect to the original value of the property, it is demanded that it is at least as secret as the write context. The label of the new value is raised to the write context. This entails that updating a property might lower the security label. Similarly,

in case the existence security label is more secret than the write context (PUT-1), it can be lowered to the write context. In the case the existence security label is not more secret than the context, execution is allowed to continue without modifying the existence label (PUT-2). The intuition is that the existence of the field is not changed. It was there before the execution of the *Put*, and it remains after.

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

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

In this rule and in the following, let $o[\sigma]$ denote an update of the structure security label. Further, we use the standard dot notation for method application, defined as follows. For pure functions, $r^\sigma.X(\overline{a}) \; E = (E[r][X](r^\sigma \cdot \overline{a}) \; E)^\sigma$, where $\overline{a}$ (the arguments) denotes a list of security labeled values, $\dot{v}$. Similarly, for functions with side-effects, $pc \vdash r^\sigma.X(\overline{a}) \; E = pc \sqcup \sigma \vdash E[r][X](r^\sigma \cdot \overline{a}) \; E$.

4.2.3. 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. Object environment records are used to inject objects into the variable environment chain: the *global object*, see Section 4.2.5, and objects originating from *with* statements.

We simplify object environment records and declarative environment records to support the same subset of operations: *HasBinding*, *GetBindingValue*, *SetMutableBinding*, *DeleteBinding*, and *ImplicitThisValue*. Of these operations, *ImplicitThisValue* deserves commenting. It is used in the semantics of function call, rule CALL in Section 4.2.9, to compute the *this* value. For object environment records, *ImplicitThisValue* returns the binding object, which ensures that methods defined on the binding object gets the binding object as the *this* value.

Object environment records Let $\mathbb{OE}$ be a family of object environment records indexed over the binding object, $\dot{r}$.

$$\begin{aligned}\mathbb{OE}(\dot{r}) = \mathbb{O}[\; &\_BindingRecord\_ \mapsto \dot{r}, \_HasBinding\_ \mapsto HasBinding, \\ &\_GetBindingValue\_ \mapsto GetBindingValue, \_SetMutableBinding\_ \mapsto SetMutableBinding, \\ &\_DeleteBinding\_ \mapsto DeleteBinding, \_ImplicitThisValue\_ \mapsto ImplicitThisValue \;]\end{aligned}$$

The algorithms for object environment records are all defined by deferring the operations to the operations of the binding object.

$$\frac{\dot{r}_2 = E[r_1][\_BindingRecord\_] \qquad \dot{b} = \dot{r}_2.\_HasProperty\_(\dot{s}) \; E}{HasBinding(r_1^\sigma, \dot{s}) \; E = \dot{b}^\sigma} \qquad \frac{\dot{r}_2 = E[r_1][\_BindingRecord\_] \qquad \dot{v} = \dot{r}_2.\_Get\_(\dot{s}) \; E}{GetBindingValue(r_1^\sigma, \dot{s}) \; E = \dot{v}^\sigma}$$

$$\frac{\dot{r}_2 = E_1[r_1][\_BindingRecord\_] \qquad E_2 = pc \vdash \dot{r}_2^\sigma.\_Put\_(\dot{s}, \dot{v}) \; E_1}{pc \vdash SetMutableBinding(r_1^\sigma, \dot{s}, \dot{v}) \; E_1 = E_2} \qquad \frac{\dot{r}_2 = E_1[r_1][\_BindingRecord\_] \qquad E_2 = pc \vdash \dot{r}_2^\sigma.\_Delete\_(\dot{s}) \; E_1}{pc \vdash DeleteBinding(r_1^\sigma, \dot{s}) \; E_1 = E_2}$$

$$\frac{\dot{r}_2 = E[r_1][\_BindingRecord\_]}{ImplicitThisValue(r_1^\sigma, \dot{s})\ E = \dot{r}_2^\sigma}$$

Thus, for object environment records, variables are stored as properties on the binding object.

The $NewObjectEnvironment$ algorithm allocates a new declarative environment and links it onto the given environment record chain.

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

Declarative environment records Let $\mathbb{DE}$ denote the declarative environment record, defined as follows.

$$\mathbb{DE} = \mathbb{O}[\ \_HasBinding\_ \mapsto HasProperty, \_GetBindingValue\_ \mapsto Get, \_SetMutableBinding\_ \mapsto Put,\\ \_DeleteBinding\_ \mapsto Delete, \_ImplicitThisValue\_ \mapsto ImplicitThisValue\ ]$$

HasProperty, *Get*, *Put*, and *Delete* are the ECMA object operations, defined in Figure 3, and $ImplicitThisValue(r_1^\sigma, \dot{s})\ \phi = undefined^\sigma$.

As before, the $NewDeclarativeEnvironment$ algorithm allocates a new declarative environment and links it onto the given environment record chain.

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

4.2.4. Variable lookup

Variable lookup is performed by *GetIdentifierReference* defined in Figure 4. *GetIdentifierReference* takes a primitive reference to the topmost variable environment and a variable name and traverses the chain of environment records (GIR-3) until the variable is found (GIR-2) or the chain ends (GIR-1). The returned result is a reference that, as mentioned in Section 4.2.1, consists of a pair formed by a security labeled primitive reference and a string. Variable lookup returns a reference $(\dot{r}, x^L)$, where x is the name of the variable, which is always public, and r is a primitive reference to the environment record containing the variable, or $undefined$, if the variable is not found. During the traversal, the security label of the access path is computed by accumulating the security labels of the traversed references in the label of the returned reference.

4.2.5. Initial environment and built-in objects

In addition to expressions and statements, the standard execution environment contains a number of built-in objects reachable from the global object, $\mathbb{G}$, defined below. As in JavaScript, the global object ends the variable environment chain via an object environment record. This means that any properties defined on the global object are available to the program as identifiers, see Section 4.2.3. In addition, the global object acts as the store for global variables, see *PutValue* in Section 4.2.8.

In the following, public existence security labels and value security labels of internal properties are frequently omitted in the case the property cannot be updated. We define a minimal initial heap containing the global object, the object constructor, $Object$, see Section 4.2.6, the object environment record for the global object, and the corresponding chaining object, the *global variable environment*.

$$\text{GIR-1}\ \frac{}{GetIdentifierReference(null^{\sigma}, x)\ E = (undefined^{\sigma}, x^{L})}$$

$$\text{GIR-2}\ \frac{\begin{array}{c} r_1 \neq null \quad r_2 = E[r_1][\_EnvironmentRecord\_] \\ true^{\sigma_2} = r_2^{\sigma_1}.\_HasBinding\_(x^{L})\ E \end{array}}{GetIdentifierReference(r_1^{\sigma_1}, x)\ E = (r_2^{\sigma_2}, x^{L})}$$

$$\text{GIR-3}\ \frac{\begin{array}{c} r_1 \neq null \quad r_2 = E[r_1][\_EnvironmentRecord\_] \\ false^{\sigma_2} = r_2^{\sigma_1}.\_HasBinding\_(x^{L})\ E \\ \dot{r}_3 = E[r_1][\_OuterEnvironment\_] \\ (\dot{r}_4, \dot{x}) = GetIdentifierReference(\dot{r}_3^{\sigma_2}, x)\ E \end{array}}{GetIdentifierReference(r_1^{\sigma_1}, x)\ E = (\dot{r}_4, \dot{x})}$$

Fig. 4. Variable lookup

$$\phi_{init} = \{\ r_G \mapsto \mathbb{G}, r_O \mapsto Object, r_E \mapsto \mathbb{OE}(r_G^L), r_V \mapsto \mathbb{C}(r_E, 0^L)\ \}$$

The initial context $\mathcal{C}_{init} = (r_G^L, r_V^L, r_V^L)$ uses the global object as the initial *this* value and sets the initial lexical environment, as well as the initial variable environment to the global variable environment. The initial environment $E_{init} = (\phi_{init}, L)$ together with the initial context form the initial execution environment.

Global object In JavaScript, the global object defines a number of properties that enriches the execution environment with constants, functions, and constructors. For brevity, we include only the object constructor, $Object$, as a constructor property of the global object, refraining from including the other constructors and prototypes.

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

4.2.6. Object objects

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 $\_Construct\_$ property. Function objects are defined in Section 4.2.7. The *ObjectConstruct* algorithm allocates a new object, initializes the $\_Prototype\_$ property to $null$, and returns the new primitive reference.

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

4.2.7. Function objects

Function objects are objects containing two internal properties, $\_Call\_$, used by function call and $\_Construct\_$, used in object construction. There are both internal function objects, e.g., the object constructor, $Object$, defined in Section 4.2.6, and user defined function objects originating from function expressions, see Section 4.2.9.

User defined function objects The function objects created when evaluating function expressions are closures storing the context (lexical environment) the function it 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 defined in the following way, where $\overline{x}$ is the list of formal parameters, c is the code of the body of the function, and $\dot{r}$ is the lexical environment the function was created in.

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

Call Calling a user defined function first allocates a new declarative environment, see Section 4.2.3, in which the arguments are bound by a process called declaration binding, defined below. Thereafter the body of the function is executed in the updated context, see the semantics of statements in Section 4.2.10. 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, σ, of the primitive reference, r_F, of the function, as is the returned value, see Section 3.1.5.

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

where $this(undefined^\sigma) = r_G^\sigma$ and $this(\dot{r}) = \dot{r}$, otherwise.

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 $undefined$.

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

$$\frac{E_2 = pc \vdash \dot{r}.\_SetMutableBinding\_(x^L, undefined^L)\ E_1}{pc, \dot{r} \vdash HoistVariables(var\ x)\ E_1 = E_2}$$

$$\frac{\begin{array}{c}E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2\end{array}}{pc, \dot{r} \vdash HoistVariables(if\ (e)\ c_1\ else\ c_2)\ E_1 = E_3}$$

$$\frac{\begin{array}{c}E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2\end{array}}{pc, \dot{r} \vdash HoistVariables(c_1; c_2)\ E_1 = E_3} \qquad \frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(while\ (e)\ c)\ E_1 = E_2}$$

$$\frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(for\ (x\ in\ e)\ c)\ E_1 = E_2}$$

$$\frac{\begin{array}{c}E_2 = pc, \dot{r} \vdash HoistVariables(c_1)\ E_1 \\ E_3 = pc, \dot{r} \vdash HoistVariables(c_2)\ E_2\end{array}}{pc, \dot{r} \vdash HoistVariables(try\ c_1\ catch(x)\ c_2)\ E_1 = E_3}$$

$$\frac{E_2 = pc, \dot{r} \vdash HoistVariables(c)\ E_1}{pc, \dot{r} \vdash HoistVariables(with\ e\ c)\ E_1 = E_2}$$

$$\frac{c \in \{throw\ e | eval\ e | skip | upg\ exc\ to\ e | upgv\ e_1\ to\ e_2 | return\ e\}}{pc, \dot{r} \vdash HoistVariables(c)\ E = E}$$

Fig. 5. Variable Hoisting

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

Variable hoisting is performed by $HoistVariables$, defined in Figure 5. Hoisting is performed by traversing the body of the function and defining all encountered variables, initializing them to $undefined$.

Construct Object construction via user defined functions uses the function as an initializer on a newly allocated ECMA object. Before passing the object, the $\_Prototype\_$ property is set to the value of the $prototype$ property 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. In JavaScript, the constructor has the option of returning a reference to an object that will be used as the instance in place of the allocated object. For simplicity, instead of having one rule for the case when the constructor function returns an object and one for when it does not, we assume that all constructor functions return an object reference. This assumption amounts to adding `return this;` at the end of all constructor functions not returning an object reference.

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

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

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

4.2.8. Auxiliary reference functions

There are two auxiliary reference functions used to interact with the target of the reference. First, *GetValue* fetches the value associated with a reference. Non-reference values are returned untouched. Subject to the limitations discussed in Section 3.1.4, an exception is raised, if the reference is undefined. Note that not only the *pc* but also the security level of the primitive reference is included in the check against the exception security level.

$$\text{GV-1}\frac{}{pc \vdash GetValue(v^\sigma)\ E = v^\sigma} \qquad \text{GV-2}\frac{\dot{v} = \dot{r}.\_Get\_(\dot{s})\ E \qquad r \neq \textit{undefined}}{pc \vdash GetValue((\dot{r}, \dot{s}))\ E = \dot{v}}$$

$$\text{GV-3}\frac{\sigma \sqcup pc <: \epsilon \qquad r = \textit{undefined}}{pc \vdash GetValue((r^\sigma, \dot{s}))\ (\phi, \epsilon) = exc\ ReferenceError^\sigma}$$

PutValue takes a reference and a value and updates the location denoted by the reference with the value. In case the reference is undefined, the update is done on the global object. This is what causes writes to undefined variables to result in a global variable being defined. Subject to the limitations discussed in Section 3.1.4, an exception is raised, if the first argument is not a reference. As for *GetValue*, the security level of the primitive reference is included in the check against the exception security level.

$$\text{PV-1}\frac{\sigma \sqcup pc <: \epsilon}{pc \vdash PutValue(v_1^\sigma, \dot{v}_2)\ (\phi, \epsilon) = (exc\ ReferenceError^\sigma, (\phi, \epsilon))}$$

$$\text{PV-2}\frac{E_2 = pc \vdash p_G^\sigma.\_Put\_(\dot{s}, \dot{v})\ E_1}{pc \vdash PutValue((undefined^\sigma, \dot{s}), \dot{v})\ E_1 = E_2}$$

$$\text{PV-3}\frac{r \neq \textit{undefined} \qquad E_2 = pc \vdash \dot{r}.\_Put\_(\dot{s}, \dot{v})\ E_1}{pc \vdash PutValue((\dot{r}, \dot{s}), \dot{v})\ E_1 = E_2}$$

4.2.9. Semantics of expressions

Let $\dot{u}$ denote exception lifted values, i.e., $\dot{u} ::= \dot{v} \mid exc\ \dot{v}$. 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 executes to $\langle \dot{u}, E_2 \rangle$ when run in E_1, the program counter security label pc, and the context $\mathcal{C}$.

The semantic rules for expressions are found in Figure 6, where most exception propagation rules have been omitted for clarity. Adding the exception propagation rules is a mechanical process where exception propagation rules should be added for each exception source. As an illustration, consider the rules for expression sequences. The two productive rules (ESEQ-1 and ESEQ-5) provide the core semantics for the evaluation of expression sequences and the other three rules (ESEQ-2, ESEQ-3, and ESEQ-4) provide exception propagation, corresponding to the following cases, respectively: 1) the first expression, e,

$$\textsc{Lit}\ \frac{}{pc, \mathcal{C} \vdash \langle l|s|n|b|undefined, E\rangle \rightarrow \langle l^L|s^L|n^L|b^L|undefined^L, E\rangle}$$

$$\textsc{Null}\ \frac{}{pc, \mathcal{C} \vdash \langle null, E\rangle \rightarrow \langle 0^L, E\rangle} \qquad \textsc{This}\ \frac{}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle this, E\rangle \rightarrow \langle \dot{\tau}, E\rangle}$$

$$\textsc{ESeq-1}\ \frac{}{pc, \mathcal{C} \vdash \langle [\,], E\rangle \rightarrow \langle [\,], E\rangle} \qquad \textsc{ESeq-2}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle exc\ \dot{v}, E_2\rangle}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle exc\ \dot{v}, E_2\rangle}$$

$$\textsc{ESeq-3}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ exc\ \dot{v}_2 = pc \vdash GetValue(\dot{v}_1)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle exc\ \dot{v}_2, E_2\rangle} \qquad \textsc{ESeq-4}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ \dot{v}_2 = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle exc\ \dot{v}_3, E_3\rangle \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle exc\ \dot{v}_3, E_3\rangle}$$

$$\textsc{ESeq-5}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}_1, E_2\rangle \\ \dot{v}_2 = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle \overline{a}, E_3\rangle \end{array}}{pc, \mathcal{C} \vdash \langle e \cdot \overline{e}, E_1\rangle \rightarrow \langle \dot{v}_2 \cdot \overline{a}, E_3\rangle} \qquad \textsc{Var}\ \frac{\dot{v} = GetIdentifierReference(\dot{le}, x)\ E}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle x, E\rangle \rightarrow \langle \dot{v}, E\rangle}$$

$$\textsc{Prj}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ \dot{r} = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ \dot{s} = pc \vdash GetValue(\dot{v}_2)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1[e_2], E_1\rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2\rangle} \qquad \textsc{Asn}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ \dot{v}_3 = pc \vdash GetValue(\dot{v}_2)\ E_2 \\ E_3 = pc \vdash PutValue(\dot{v}_1, \dot{v}_3)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1 = e_2, E_1\rangle \rightarrow \langle \dot{v}_3, E_3\rangle}$$

$$\textsc{EDel-1}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2\rangle \\ IsPropertyReference((r, s))\ E_2 \\ (\dot{v}, E_3) = pc \vdash \dot{r}.\_Delete\_(\dot{s})\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \rightarrow \langle \dot{v}, E_3\rangle} \qquad \textsc{EDel-2}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle (undefined^\sigma, \dot{s}), E_2\rangle}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \rightarrow \langle true^L, E_2\rangle}$$

$$\textsc{EDel-3}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2\rangle \\ \neg IsPropertyReference((r, s))\ E_2 \\ (\dot{v}, E_3) = pc \vdash \dot{r}.\_DeleteBinding\_(\dot{s})\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle delete\ e, E_1\rangle \rightarrow \langle \dot{v}, E_3\rangle}$$

$$\textsc{Call}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}, E_2\rangle \qquad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle \overline{a}, E_3\rangle \\ (\dot{u}, E_4) = pc \vdash E_3[r][\_Call\_](this(\dot{v})\ E_3, \dot{r}, \overline{a})\ E_3 \end{array}}{pc, \mathcal{C} \vdash \langle e(\overline{e}), E_1\rangle \rightarrow \langle \dot{u}, E_4\rangle}$$

$$\textsc{New}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle \dot{v}, E_2\rangle \qquad \dot{r} = pc \vdash GetValue(\dot{v})\ E_2 \\ pc, \mathcal{C} \vdash \langle \overline{e}, E_2\rangle \rightarrow \langle \overline{a}, E_3\rangle \\ (\dot{u}, E_4) = pc \vdash E_3[r][\_Construct\_](\dot{r}, \overline{a})\ E_3 \end{array}}{pc, \mathcal{C} \vdash \langle new\ e(\overline{e}), E_1\rangle \rightarrow \langle \dot{u}, E_4\rangle}$$

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

$$\textsc{BiOp}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \rightarrow \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ v_3^{\sigma_1} = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ v_4^{\sigma_2} = pc \vdash GetValue(\dot{v}_2)\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle e_1 \star e_2, E_1\rangle \rightarrow \langle (v_3 \star v_4)^{\sigma_1 \sqcup \sigma_2}, E_2\rangle} \qquad \textsc{Type-1}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle v^\sigma, E_2\rangle}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \rightarrow \langle typeof(v)^\sigma, E_2\rangle}$$

$$\textsc{Type-2}\ \frac{\begin{array}{c} pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle (\dot{r}, \dot{s}), E_2\rangle \\ r \neq undefined \qquad v^\sigma = GetValue((\dot{r}, \dot{s}))\ E_2 \end{array}}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \rightarrow \langle typeof(v)^\sigma, E_2\rangle} \qquad \textsc{Type-3}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \rightarrow \langle (undefined^\sigma, \dot{s}), E_2\rangle}{pc, \mathcal{C} \vdash \langle typeof(e), E_1\rangle \rightarrow \langle undefined^\sigma, E_2\rangle}$$

Fig. 6. Semantics of expressions

results in an exception, 2) the call to $GetValue$ results in an exception, and 3) one of the expressions in the tail of the sequence, $\overline{e}$, results in an exception, respectively. All of these cases terminate the evaluation by propagating the exception.

The expression rules are formulated in terms of the primitives of the previous sections. The string s, numbers, n, booleans, b, and *undefined* literals evaluate to their semantic counterpart. They are labeled L, whereas the label literals are labeled with themselves (LIT). This allows the label literals to be used as label constants in the upgrade instructions, see Section 4.2.11.

The *null* literal evaluates to the null primitive reference 0 (NULL), and the *this* literal evaluates to the current this primitive reference $\dot{\tau}$ (THIS).

Expression sequences (ESEQ-1 and ESEQ-5) are used by function call, object construction, projection, assignment, and binary operators. They are evaluated in order, resulting in a sequence of values.

Identifier dereference (VAR) uses $GetIdentifierReference$. All rules that contain evaluation of subexpressions use $GetValue$ to convert the results to values apart from assignment (ASN) that uses $PutValue$ to update the location denoted by the reference.

Delete uses $Delete$ (EDEL-1) or $DeleteBinding$ (EDEL-3) depending on if the target is an object or an environment record (indicated by $IsPropertyReference$). Attempting to delete an undefined variable does not have any effect (EDEL-2).

Function creation (FUNC) 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 call (CALL) uses $Call$. Object creation (NEW) uses $Construct$.

Note the $this(\dot{v})\ E_3$ of the function call, which computes the *this* binding of the invocation, from the reference $\dot{v} = (r, s)^{\sigma}$. It is computed as follows: 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, notice how *typeof* (TYPE-3) 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. Thus, *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.

4.2.10. Semantics of statements

Let $R ::= E \mid \langle \dot{u}, E \rangle$, where E indicates normal termination without return value, and $\langle \dot{u}, E \rangle$ indicates either exceptional termination or termination with a return value, depending on whether $\dot{u}$ is $exc\ \dot{v}$ or $\dot{v}$. The semantics of statements is of the form $pc, \mathcal{C} \vdash \langle c, E \rangle \rightarrow R$, read as c executes to R when run in E, program counter security label pc, and context $\mathcal{C}$.

Figure 7 displays the semantic rules of statements, where the rules for exception propagation have been omitted for clarity. Sequence (SEQ-1), iteration in the form of *while* (WHILE), and conditional choice (IF-1 and IF-2) are all standard. The conditional choice raises the security context of the chosen branch to the security label of the controlling expression, and *while* is formulated in terms of conditional choice. The *for-in* statement (FOR-IN) iterates over the external properties of an object. For each external property name in the object, the given variable is updated with the name using the existence security label of the property as the security label, and the body of the *for-in* is run. The iteration of *for-in* is provided by the three rules (ITER-1, ITER-2, and ITER-3) 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 referenced by $\dot{v}$ to $\dot{s}$ before running c. Note that c is run in the context of the

$$\textsc{Seq-1}\ \frac{pc, \mathcal{C} \vdash \langle c_1, E_1\rangle \to E_2 \quad pc, \mathcal{C} \vdash \langle c_2, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle c_1; c_2, E_1\rangle \to E_3} \qquad \textsc{If-1}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle true^{\sigma}, E_2\rangle \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c_1, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2, E_1\rangle \to E_3}$$

$$\textsc{If-2}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle false^{\sigma}, E_2\rangle \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c_2, E_2\rangle \to E_3}{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ c_1\ \mathit{else}\ c_2, E_1\rangle \to E_3}$$

$$\textsc{While}\ \frac{pc, \mathcal{C} \vdash \langle \mathit{if}\ (e)\ \{c; \mathit{while}\ (e)\ c\}\ \mathit{else}\ \mathit{skip}, E_1\rangle \to E_2}{pc, \mathcal{C} \vdash \langle \mathit{while}\ (e)\ c, E_1\rangle \to E_2}$$

$$\textsc{Iter-1}\ \frac{}{pc, \mathcal{C} \vdash \langle([\,], \dot{v}, c), E\rangle \to E} \qquad \textsc{Iter-3}\ \frac{\neg IsExternal(s) \quad pc, \mathcal{C} \vdash \langle(\overline{s}, \dot{v}, c), E_1\rangle \to E_2}{pc, \mathcal{C} \vdash \langle(s^{\sigma} \cdot \overline{s}, \dot{v}, c), E_1\rangle \to E_2}$$

$$\textsc{Iter-2}\ \frac{IsExternal(s) \quad E_2 = pc \sqcup \sigma \vdash PutValue(\dot{v}, s^{\sigma})\ E_1 \quad pc \sqcup \sigma, \mathcal{C} \vdash \langle c, E_2\rangle \to E_3 \quad pc, \mathcal{C} \vdash \langle(\overline{s}, \dot{v}, c), E_3\rangle \to E_4}{pc, \mathcal{C} \vdash \langle(s^{\sigma} \cdot \overline{s}, \dot{v}, c), E_1\rangle \to E_4}$$

$$\textsc{Ret}\ \frac{pc, \mathcal{C} \vdash \langle e, E_1\rangle \to \langle \dot{v}, E_2\rangle \quad \dot{u} = pc \vdash GetValue(\dot{v})\ E_2}{pc, \mathcal{C} \vdash \langle return\ e, E_1\rangle \to \langle \dot{u}, E_2\rangle}$$

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

$$\textsc{Try-1}\ \frac{pc, \mathcal{C} \vdash \langle c_1, (\phi_1, \epsilon_1)\rangle \to (\phi_2, \epsilon_2)}{pc, \mathcal{C} \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1)\rangle \to (\phi_2, \epsilon_1)}$$

$$\textsc{Try-2}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle c_1, (\phi_1, \epsilon_1)\rangle \to \langle exc\ \dot{v}, E_2\rangle \\ (le_2^L, E_3) = pc \vdash NewDeclarativeEnvironment(\dot{le}_1)\ E_2 \\ \dot{r} = E_3[le_2][\_EnvironmentRecord\_] \\ (\phi_4, \epsilon_4) = \dot{r}.\_SetMutableBinding\_(x^L, \dot{v})\ E_3 \\ pc \sqcup \epsilon_4, (\dot{\tau}, le_2^L, \dot{ve}) \vdash \langle c_2, (\phi_4, \epsilon_1)\rangle \to E_5 \end{array}}{pc, (\dot{\tau}, \dot{le}_1, \dot{ve}) \vdash \langle try\ c_1\ catch(x)\ c_2, (\phi_1, \epsilon_1)\rangle \to E_5}$$

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

$$\textsc{Eval}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle e, E_1\rangle \to \langle s^{\sigma}, E_2\rangle \quad c = parse(s) \\ E_3 = pc \sqcup \sigma, \dot{ve} \vdash HoistVariables(c)\ E_2 \\ pc \sqcup \sigma, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle c, E_3\rangle \to E_4 \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle eval\ e, E_1\rangle \to E_4}$$

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

Fig. 7. Semantics of statements

label of the corresponding string (ITER-2), and that internal properties are skipped (ITER-3). The former might seem counterintuitive. In the light of the operation of an ordinary for loop, one might expect an accumulated security content. However, this is not necessary. The reason for this is that there is no way for executions of the body corresponding to properties with secret existence labels (the secret executions) to communicate anything public to later executions corresponding to properties with public existence labels (the public executions), since the secret executions are under secret control. From the perspective of the public executions, it is not possible to detect if or how they are interleaved with secret executions without raising their own security context. Consider the following program, where the property q with public existence may or may not be preceded by the property p with secret existence. The program tries to communicate this fact by accumulating into `i` how many times the body of the *for-in* is executed and storing into `l` if property q was first or not.

```
1 o = upgs {} to h;
2 if (h) { o.p = 'a'; }
3 o.q = 'b'; i = 0; l = true;
4 for (x in o) {
5   if (x === 'q' && i === 0) { l = false; }
6   i += 1;
7 }
```

First, the structure of `o` must be upgraded to allow for the addition of property p under secret control. Consider the two possible execution of the above program. If `h` is false, property p will not be added to `o`, and the first execution of the body (corresponding to the property q) will update `l` to `false`. If, on the other hand, `h` is true, property p will be added to `o`. Since the existence security label of p is secret, the first execution (corresponding to the property p) will be under secret control, and the execution will be stopped when trying to update `i` on line 6, which is public. In the above terms, the secret execution was prevented to communicate a public value.

Now, consider what happens if we modify the example so that `i` is secret. If `h` is false, property p will not be added to `o`, but now, when the first execution (corresponding to the property q) tries to update `l` the execution will fail with a security error. The reason for this is, by making `i` secret we have made the inner conditional on line 5 secret, which prevents any public updates. If, on the other hand, `h` is true, property p will be added to `o` and `l` will remain false. In the above terms, any differences detected by later public executions must be secret and will cause the public execution to enter a secret context.

Exception throwing (THROW) is constrained by the demand that the security context pc is below the exception security label ϵ, i.e., no low exceptions are allowed in secret context.

The execution of *try-catch* is divided into normal (TRY-1) and exceptional (TRY-2) 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 local to the handler, but escape to the outer variable environment, similar to variables declared in the bodies of *eval* and *with* described below. 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

$$\textsc{Upg}\ \frac{pc, \mathcal{C} \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \quad \dot{v}_3 = pc \vdash GetValue(\dot{v}_1)\ E_2 \quad v_4^\sigma = pc \vdash GetValue(\dot{v}_2)\ E_2}{pc, \mathcal{C} \vdash \langle upg\ e_1\ to\ e_2, E\rangle \to \langle \dot{v}_3^\sigma, E_2\rangle}$$

$$\textsc{UpgExc}\ \frac{pc <: \epsilon \quad pc, \mathcal{C} \vdash \langle e, E\rangle \to \langle \dot{v}_1, (\phi, \epsilon)\rangle \quad v_2^\sigma = pc \vdash GetValue(\dot{v}_1)\ (\phi, \epsilon)}{pc, \mathcal{C} \vdash \langle upg\ exc\ to\ e, E\rangle \to (\phi, \epsilon \sqcup \sigma)}$$

$$\textsc{UpgS}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [e_1, e_2], E_1\rangle \to \langle [\dot{v}_1, \dot{v}_2], E_2\rangle \\ r^{\sigma_1} = pc \vdash GetValue(\dot{v}_1)\ E_2 \qquad v_3^{\sigma_2} = pc \vdash GetValue(\dot{v}_2)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \quad o_1 = E_2[r] \quad o_1 = \{\dots, \sigma_3\} \quad \sigma <: \sigma_3 \quad o_2 = o_1[\sigma \sqcup \sigma_2] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle upgs\ e_1\ to\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

$$\textsc{UpgV}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [x, e_2], E_1\rangle \to \langle [(r^{\sigma_1}, \dot{s}), \dot{v}_1], E_2\rangle \quad v_2^{\sigma_2} = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \quad o_1 = E_2[r] \quad o_1 = \{\dots, \sigma_3\} \quad \sigma <: \sigma_3 \quad o_2 = o_1[\sigma \sqcup \sigma_2] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle upgv\ x\ to\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

$$\textsc{UpgE}\ \frac{\begin{array}{c} pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle [e_1, e_2], E_1\rangle \to \langle [(r^{\sigma_1}, s^{\sigma_2}), \dot{v}_1], E_2\rangle \qquad v_2^{\sigma_3} = pc \vdash GetValue(\dot{v}_1)\ E_2 \\ \sigma = pc \sqcup \epsilon \sqcup \sigma_1 \sqcup \sigma_2 \quad o_1 = E_2[r] \quad o_1 = \{s \overset{\sigma_4}{\mapsto} \dot{v}_3, \dots\} \quad \sigma <: \sigma_4 \quad o_2 = o_1[s \overset{\sigma \sqcup \sigma_3}{\mapsto} \dot{v}_3] \end{array}}{pc, (\dot{\tau}, \dot{le}, \dot{ve}) \vdash \langle upge\ e_1\ to\ e_2, E\rangle \to E_2[r \mapsto o_2]}$$

Fig. 8. Upgrade semantics

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 label, since any (uncaught) exceptions in the handler escape the *try-catch*.

The return statement (RET) takes an expression and makes the corresponding value the return value of the function. Note the simplifying assumption that all functions have a unique exit point in terms of a return statement. This is manifested in that the function call expression (CALL) demands that a value is returned, while the statement sequence (SEQ-1 together with the rules for exception propagation) does not propagate return. In order to support unconstrained use of return a *return security label* can be introduced, serving a similar purpose as the exception security label. Since the labels are closely related, we have opted to let the exception security label be representative in the formal development. Naturally, in the full implementation this restriction and other restrictions of the formal development have been lifted.

The *eval* statement (EVAL) 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 (WITH) injects an object into the lexical environment chain. Properties of the injected object shadows existing variables for both reading and writing.

4.2.11. Upgrade instructions

Figure 8 defines one upgrade expression and four upgrade statements. All of the upgrade instructions are dynamic, in the sense that the target label is taken from the reduction of the second argument. This allows upgrading of potential write targets to match the control structure of the program without enforcing specific labels. For example, in the following snippet, l is upgraded to the label of h:

```
l = upg l to h;
if (h) {
```

```
    l = 1;
}
```

The upgrade expression (UPG) upgrades the result of the first parameter to the label of the second. Upgrading to specific labels is made possible by the introduction of label literals that carry their own label, see Section 4.2.9.

The upgrade exception label statement (UPGEXC) upgrades the exception label to the label of the parameter, provided the upgrade can be allowed by the security context. The upgrade structure label statement (UPGS) upgrades the structure of the object referred to by the first parameter to the label of the second, provided that the write context of the object allows the upgrade. Similarly, the upgrade structure label of environment records statement (UPGV) upgrades the structure of the environment record of the given variable x to the label of the second parameter, provided that the write context of the environment record allows the upgrade. Finally, the upgrade existence label statement (UPGE) upgrades the existence label of the property referred to by the first parameter to the security label of the second parameter, provided that the write context of the property allows it. In all upgrade statements, the write context is taken into the account when creating the resulting label.

5. Information-flow security and transparency

A common policy for information-flow security is noninterference [19,29]. Noninterference can be formally framed as the preservation of a family of *low-equivalence* relations, denoted $\sim$, under execution. As is standard for languages with references [7], the family is indexed by a relation, denoted 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* [72,63] noninterference: two programs are considered *noninterfering* if all terminating runs for the same public input agree on all 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.

5.1. Low-equivalence

Noninterference is formulated in terms of a family of low-equivalence relations 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 the values are equal.

Figure 9 defines the family of relations, indexed by a relation on primitive references β. This relation represents a bijection between the low-reachable parts of the heaps, i.e, all locations that can be reached by public access paths.

First, let $dom_L(o)$ denote the public domain of o, i.e., the set of all properties with public existence. Let $\varsigma(o)$ denote the structure security label of o. The sets V and R range over labeled native values and references, respectively. The bijection β is forced to contain at least all low-reachable references by the interaction between LE-PR-L and LE-H. For any pair of primitive references $(r_1, r_2) \in \beta$, LE-H demands that the corresponding objects are low-equivalent. In turn, LE-O-L and LE-O-H demand

$$\text{LE-PR-L}\frac{r_1\ \beta\ r_2}{r_1^L \sim_\beta r_2^L} \qquad \text{LE-V-L}\frac{v \notin R}{v^L \sim_\beta v^L} \qquad \text{LE-V-H}\frac{}{v_1^H \sim_\beta v_2^H} \qquad \text{LE-R}\frac{\dot{r}_1 \sim_\beta \dot{r}_2 \qquad \dot{s}_1 \sim_\beta \dot{s}_2}{(\dot{r}_1, \dot{s}_1) \sim_\beta (\dot{r}_2, \dot{s}_2)}$$

$$\text{LE-IP}\frac{p \notin V}{p \sim_\beta p} \qquad \text{LE-VS1}\frac{\dot{v}_1 \sim_\beta \dot{v}_2 \qquad \overline{v}_1 \sim_\beta \overline{v}_2}{\dot{v}_1 \cdot \overline{v}_1 \sim_\beta \dot{v}_2 \cdot \overline{v}_2} \qquad \text{LE-VS2}\frac{}{[\,] \sim_\beta [\,]}$$

$$\text{LE-EV-L1}\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{exc\ \dot{v}_1 \sim_{\beta,L} exc\ \dot{v}_2} \qquad \text{LE-EV-L2}\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{\dot{v}_1 \sim_{\beta,L} \dot{v}_2} \qquad \text{LE-EV-H}\frac{}{\dot{u}_1 \sim_{\beta,H} \dot{u}_2}$$

$$\text{LE-O-L}\frac{\begin{array}{c}\varsigma(o_1) = \varsigma(o_2) = L \\ dom(o_1) = dom(o_2) \qquad dom_L(o_1) = dom_L(o_2) \\ (s \overset{L}{\mapsto} \dot{p}_1) \in o_1 \wedge (s \overset{L}{\mapsto} \dot{p}_2) \in o_2 \Rightarrow \dot{p}_1 \sim_\beta \dot{p}_2\end{array}}{o_1 \sim_\beta o_2} \qquad \text{LE-H}\frac{r_1\ \beta\ r_2\ \Rightarrow\ \phi_1[r_1] \sim_\beta \phi_2[r_2]}{\phi_1 \sim_\beta \phi_2}$$

$$\text{LE-O-H}\frac{\begin{array}{c}\varsigma(o_1) = \varsigma(o_2) = H \qquad dom_L(o_1) = dom_L(o_2) \\ (s \overset{L}{\mapsto} \dot{p}_1) \in o_1 \wedge (s \overset{L}{\mapsto} \dot{p}_2) \in o_2 \Rightarrow \dot{p}_1 \sim_\beta \dot{p}_2\end{array}}{o_1 \sim_\beta o_2}$$

$$\text{LE-Ctx}\frac{\dot{\tau}_1 \sim_\beta \dot{\tau}_2 \qquad \dot{le}_1 \sim_\beta \dot{le}_2 \qquad \dot{ve}_1 \sim_\beta \dot{ve}_2}{(\dot{\tau}_1, \dot{le}_1, \dot{ve}_1) \sim_\beta (\dot{\tau}_2, \dot{le}_2, \dot{ve}_2)} \qquad \text{LE-Env}\frac{\phi_1 \sim_\beta \phi_2}{(\phi_1, \epsilon) \sim_{\beta,\epsilon} (\phi_2, \epsilon)}$$

$$\text{LE-EE}\frac{\dot{u}_1 \sim_{\beta,\epsilon} \dot{u}_2 \qquad E_1 \sim_{\beta,\epsilon} E_2}{\langle \dot{u}_1, E_1 \rangle \sim_{\beta,\epsilon} \langle \dot{u}_2, E_2 \rangle} \qquad \text{LE-EE-H1}\frac{E_1 \sim_{\beta,H} E_2}{E_1 \sim_{\beta,H} \langle \dot{u}_2, E_2 \rangle} \qquad \text{LE-EE-H2}\frac{E_1 \sim_{\beta,H} E_2}{\langle \dot{u}_1, E_1 \rangle \sim_{\beta,H} E_2}$$

$$\text{LE-PD-L1}\frac{\dot{v}_1 \sim_\beta \dot{v}_2}{\{value \mapsto \dot{v}_1\} \sim_\beta \{value \mapsto \dot{v}_2\}} \qquad \text{LE-PD-L2}\frac{}{undefined^\sigma \sim_\beta undefined^\sigma}$$

$$\text{LE-PD-H1}\frac{}{\{value \mapsto \dot{v}^H\} \sim_\beta undefined^H} \qquad \text{LE-PD-H2}\frac{}{undefined^H \sim_\beta \{value \mapsto \dot{v}^H\}}$$

$$\text{LE-ER}\frac{r_{11}\ \beta\ r_{12} \qquad \dot{r}_{21} \sim_\beta \dot{r}_{22}}{\{\_Env...Record\_ \mapsto r_{11}, \_OuterEnv..._ \mapsto \dot{r}_{21}\} \sim_\beta \{\_Env...Record\_ \mapsto r_{12}, \_OuterEnv..._ \mapsto \dot{r}_{22}\}}$$

Fig. 9. Low-equivalence

that the low domain of the objects are equal and that all corresponding values are low-equivalent. Thus, any public primitive references contained in these properties are forced to be in β by LE-PR-L. This guarantees that the corresponding objects are low-equivalent. The process enforces a closure property on β: for any pair of primitive references (r_1, r_2) in β, all pairs of low-reachable references (w.r.t. the same access path) will also be in β. In addition, for objects with public structure LE-O-L demands that the public domains are equal in addition to the demand that the public domains are equal.

On the top level, the process is initiated by LE-Ctx that via LE-ER demands that the primitive references corresponding to the topmost lexical and variable environments are in β. This injects at least one pair of primitive references into β carried over to LE-H via LE-Env. This pair provides the root for the low-reachable subheap of each environment, ensuring that they are isomorphic and that corresponding public primitive non-references are equal.

For exception lifted values, LE-EV-L1 and LE-EV-L2 guarantee that the cause of exceptions is independent of secrets unless the exception security label is secret. Further exception-related rules are LE-EE via LE-Env, where the latter exposes the exception security label as a decoration on the low-equivalence relation. This carries the exception security label to the exception lifted values. Regardless of the value of the exception security label, it is demanded that the environments are low-equivalent. This corresponds

to the intuition that an exception is not able to modify the environment, only affect the control flow. As for all security labels, the exception security label is forced to be equal in both environments.

5.2. 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:

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

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. By strong induction on the height of the derivation tree. Since the definitions of statements, expressions, $FunctionCall$, and $FunctionConstruct$ are mutually recursive, we strengthen the induction predicate to the conjunction of the noninterference for the respective constructs. We outline of the main components of the proof. The full proof can be found in the extended version of this paper [35].

For convenience, expression sequences and their construction are lifted into expressions for the proof. As is standard, the noninterference theorem uses a supporting theorem, confinement (Theorem 2), that proves freedom of public side effects under secret control. Since the semantics is frequently formulated in terms of primitive constructions, like ECMA objects and their operations, many of the cases rely only on confinement and noninterference of the primitive constructions. A selection of notable exceptions among the expressions is presented below. The proof explanations are intended to give a broad picture and convey the intuition. The discussion is rooted in comparing the two derivation trees representing the execution, which is manifested as a case analysis of the last applied derivation rule, under the assumption that the the proof holds for all sub derivations of expression and statement execution.

Expression sequence Expression sequence serves as an illustration of exception handling, which is omitted in the remaining proof cases. There are two rules for successful execution: ESEQ-1 and ESEQ-5. The other three rules are for exception propagation, giving a total of five rules. This gives 12 proof cases when symmetry is taken into account (25 otherwise). First, the cases where both derivations end with the same rule correspond to the situations, where both executions terminate normally or both executions throw exceptions, and are easily established by the induction hypothesis. The remaining cases can be easily dismissed as impossible. Combining ESEQ-1 with any of the other rules in not possible, since both executions work on the same sequence of expressions. The other cases contradict the induction hypothesis.

Function call The proof is divided into two cases, based on the security label of the primitive references identifying the object that represents the function.

1) If the security label is public, then the objects are low-equivalent, which, in particular, entails that the algorithms for the internal property _$Call$_ must be equal. In fact, they must be equal to $FunctionCall$ (there is only one possibility). The result follows from the induction hypothesis.

Object construction The proof is similar to the proof for function call, with the difference that there are now two possible algorithms for the internal property $\_Construct\_$. The difference is, however, minimal. In the first case, we still have that the algorithms must be equal. We get two subproofs: one for $ObjectConstruct$ and one for $FunctionConstruct$. The former follows from a supporting lemma, and the latter follows from the induction hypothesis.

For statements, the following proof cases are representative for the other proof cases.

Conditional The conditional statement has two productive rules, which gives three cases, when symmetry is taken into account. Each case is split into two subproofs based on the security label of the controlling expression. 1) For a public label corresponding to public control flow, the result follows from the induction hypothesis. 2) For a secret label, the proof proceeds by confinement together with symmetry and transitivity of the low-equivalence relation. Regardless of the paths taken through the conditional, confinement allows us to establish the downward equivalences. In turn, reflexivity and transitivity allow us to conclude:

$$E_{21} \sim_{op(fst(\beta)),\epsilon} E_{11} \wedge E_{11} \sim_{\beta,\epsilon} E_{12} \wedge E_{12} \sim_{snd(\beta),\epsilon} E_{22} \Rightarrow E_{21} \sim_{\beta,\epsilon} E_{22}$$

The use of $op(\beta)$ for opposite relation of β, $fst(\beta)$, and $snd(\beta)$ for the identity relation induced by the domain and the codomain of β respectively is a technicality pertaining to the reflexivity and transitivity of the low-equivalence relation. The four proof cases are illustrated below, where the dashed lines represent executions for which confinement is used to establish low-equivalence between the initial and the final environment.

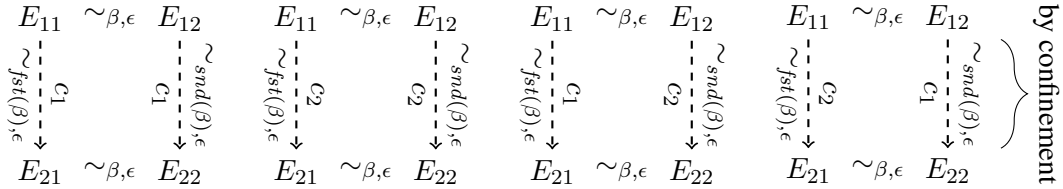

Fig. 10. From left, illustration of (IF-1, IF-1), (IF-2, IF-2), (IF-1, IF-2), and (IF-2, IF-1)

Try-catch The proof for try-catch follows the same intuition with respect to confinement as the proof for the conditional. We have two rules which give three cases, when symmetry is taken into account. 1) If no exceptions occur in the body of the try-catch, the result follows from the induction hypothesis, as illustrated by the leftmost picture below. 2) If exceptions occur in the body of the try-catch in both executions, then we have two subcases, depending of the exception security label. In case it is public, the result follows from the induction hypothesis. Otherwise, the result follows from confinement, analogously to the conditional above. 3) If an exception occurs in the body of the try-catch in only one of the executions, the induction hypothesis gives that the exception security label must be secret. The result follows from confinement and transitivity of the low-equivalence relation. This case is illustrated in the second figure below.

□

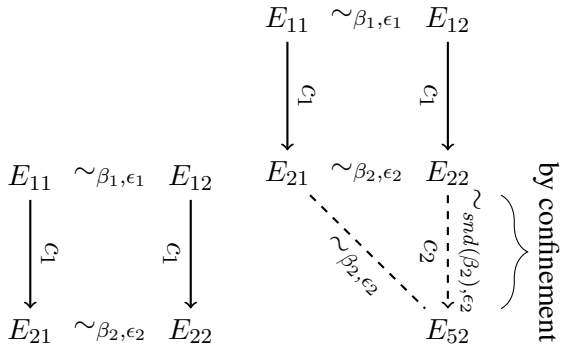

Fig. 11. From left, illustration of (TRY-1, TRY-1) and (TRY-1, TRY-2)

5.2.1. Noninterference of primitive constructions

With respect to noninterference of primitive constructions, there are two major categories: pure and impure primitive constructions. For pure primitive constructions, the label of the result is the least upper bound of the security labels of all used values. An interesting special case of this is constructions that read variables of properties, since these constructions embody the notion of read context introduced above. As an illustration consider the following outline of the proof of noninterference of $GetIdentifierReference$.

Lemma 1. *Noninterference of* $GetIdenfierReference$.

$$E_1 \sim_\beta E_2 \ \wedge\ \dot{r}_{11} \sim_\beta \dot{r}_{12} \wedge$$
$$GetIdentifierReference(\dot{r}_{11}, x)\ E_1 = (\dot{r}_{21}, x^L) \wedge$$
$$GetIdentifierReference(\dot{r}_{12}, x)\ E_2 = (\dot{r}_{22}, x^L) \Rightarrow \dot{r}_{21} \sim_\beta \dot{r}_{22}$$

Proof. The proof proceeds by induction on the height of the derivation tree. There are three rules, which gives six cases, when symmetry is taken into account. The intuition behind the rules is:

1. GIR-1 the base case, where the variable was not found, and the end of the chain has been reached,
2. GIR-2 the base case, where the variable was found, and
3. GIR-3 the recursive case, where the variable has not yet been found, but the end of the chain has not yet been reached.

In all cases, the security label of the incoming reference is used to taint the result. In 1) and 2) this corresponds to the immediate read context, while in 3) it corresponds to the accumulated read context that is added to the read context resulting from the recursive call. The proof consists of two parts:

Part one: The first three proof cases apply when the derivation trees (at least initially) share the same structure, i.e., that both derivations end with the same rule. The proofs for these cases are similar. If the incoming reference is public, the result follows from noninterference of the use primitives (together with the induction hypothesis in case 3). If the incoming reference is secret, then the result must also be secret, which places no further demands on the result.

Part two: The last three proof cases apply when the derivation trees differ. As an example, this may occur if the variable was found in one environment but not the other. The proofs for all these cases are based on showing that the result must be secret. The three cases are as follows.

1. GIR-1 together with GIR-2 corresponding to the case when the variable is not found in the environment in one execution and that it is found in the other. The fact that the pointer to the environment record is $null$ in one execution and not the other means that the label of the reference must be secret, and thus that the label of the result is secret.
2. GIR-1 together with GIR-3 corresponding to the case when the variable is not found in the environment in one execution and that it is not found yet in the other. The proof is the same as in the previous case.
3. GIR-2 together with GIR-3 corresponding to the case when the variable is found in the environment in one execution and that it is not found yet in the other. We get two cases. If the label of the incoming reference is secret, then the result is secret, and we can conclude. Otherwise, since the variable is found in one execution but not the other, this entails that the result of $\_HasBinding\_$ must be secret, and the result follows.

□

For impure primitive constructions, the write context is taken into account for all side effects. Examples of impure primitive include Put and $Delete$. To illustrate, consider the following outline for the proof on noninterference of Put.

Lemma 2. *Noninterference of Put.*

$$E_{11} \sim_\beta E_{21} \ \wedge \ \dot{r}_1 \sim_\beta \dot{r}_2 \ \wedge \ \dot{s}_1 \sim_\beta \dot{s}_2 \ \wedge \ \dot{v}_1 \sim_\beta \dot{v}_2 \wedge$$
$$Put(\dot{r}_1, \dot{s}_1, \dot{v}_1)\ E_{11} = E_{12} \ \wedge Put(\dot{r}_2, \dot{s}_2, \dot{v}_2)\ E_{21} = E_{22} \Rightarrow E_{12} \sim_\beta E_{22}$$

Proof. There are three rules, which gives six cases, when symmetry is taken into account. The intuition behind each of the rules is:

1. PUT-1 The property already exists, and it is made sure that label of the write context (the write path together with the security context) is not above the label of the target. In addition, the existence label of the property was below the write context, and it is left untouched.
2. PUT-2 The property already exists, and it is made sure that label of the write context (the write path together with the security context) is not above the label of the target. In addition, the existence label is more secret than the write context and is lowered to the write context.
3. PUT-3 The property does not exist. In this case, the structure of the object is affected by the write, and it is made sure that the write context is not above the structure security label.

The proof proceeds depending on the security label of the write path, i.e., the join of the security label of the reference and the security label of the property name. It can be divided into two parts. For public write paths, we know that the update objects are low-equivalent. We have to show that the update retains the low-equivalence. For secret write paths, we do not necessarily know that the updated objects are low-equivalent. They may, however, be low-equivalent with other objects on the heap. Thus, in such case we must show that any such potential low-equivalences are retained by the update.

Part one: The simplest three proof cases consist of the cases when the same rules are applied.

1. PUT-1 together with PUT-1. The property exists in both executions, and neither the existence label nor the structure label is touched. In case the write path is public, identical modifications are done to low-equivalent object, and low-equivalence is preserved. Otherwise, the labels of the targets are

secret (and remains secret), and the update preserves any low-equivalences the updated objects may have been in.
2. PUT-2 together with PUT-2. Analogous to the above case, with the additional update of the existence security label.
3. PUT-3 together with PUT-3. The property does not exist in either of the executions. Thus, there are no previous values to place demands on, but the structure of the object is changed. In case the write path is public, identical additions are done to low-equivalent objects, and low-equivalence is preserved. Otherwise, the structure security label of the objects is secret, and the new properties are added with secret existence security labels. Since properties with secret existence label are not considered by the definition of low-equivalence of objects, the update preserves any low-equivalences the updated objects may have been in.

Part two: The second part of the proof is more interesting as it corresponds to the case when the updates behave differently with respect to the objects. The general intuition is that for secret write paths the targets have to be secret, and for public write paths the low-equivalence of the updated objects ensure that any position where the objects differ must be secret.

1. PUT-1 together with PUT-2. This case corresponds to the case where the property exists, but with different existence security labels, one public and one secret. For a public write path, this is not possible (low-equivalence ensures that the same property has the same existence security label in both objects). For a secret write path, this is possible, since different properties may be updated in the different objects. In both cases. the existence label remains untouched. Similar to the first two cases, in the first part of the proof, the labels of the targets the labels of the targets are secret (and remains secret), and the update preserves any low-equivalences the updated objects may have been in.
2. PUT-1 together with PUT-3. This case corresponds to the case where the property exists in one of the executions but not the other. For a public write path, the low-equivalence of the updated object entails that the structure security label must be secret (otherwise the property would have been in both objects). In this case, identical modifications are done to low-equivalent object, and low-equivalence is preserved. A secret write path ensures that the structure security labels of the objects are secret as are the existence security label of existing property and the security label of the value of the property. The update of this object does not change any of the existing security labels, and the update preserves any low-equivalence the updated object may have been in. Similarly, adding a property with secret existence label to an object with secret structure security label retains any low-equivalence the updated object may have been in.
3. PUT-2 together with PUT-3. This case corresponds to the case where the property exists and is not below the existence security label in one of the executions but not the other. For public write paths, the proof corresponds to the first case, and for secret write paths, the second case above.

□

The full proofs including all supporting lemmas can be found in the extended version of this paper [35].

5.3. Confinement

Confinement expresses that execution under secret control does not modify any public labels. We express this by demanding that the initial and the final environments are low-equivalent. This formulation

makes it easy to use confinement together with symmetry and transitivity of low-equivalence in the proof of noninterference. See Section 5.2 for an explanation and illustration of this.

Theorem 2. *Confinement of statements*

$$\forall c \,.\, conf(c)$$

where $pc \sqcup \epsilon = H \wedge E_1 \sim_{\beta,\epsilon} E_1 \wedge pc, \mathcal{C} \vdash \langle c, E_1\rangle \rightarrow E_2 \Rightarrow E_1 \sim_{\beta,\epsilon} E_2$

Proof. The proof follows the overall structure of the proof of noninterference, in that it utilizes a strengthened induction predicate, and lifts expression sequences and the iteration support into the expressions. For confinement, only the constructions that have side effects are interesting. The proofs follow immediately from the fact that all rules with side effects demand that the security context is below the write target. □

The full proofs including all supporting lemmas can be found in the extended version of this paper [35].

5.4. 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 execution 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 3 (Transparency)**.** *It holds that*

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

Proof. Immediate from inspection of the rules. From a progress perspective, the rules for $Delete$, Put, $GetValue$, $PutValue$, $throw$, and *upg* instructions may stop the execution. No instrumented rules add possibilities of execution that are not present in the un-instrumented semantics. □

6. From theory to practice of information-flow security of JavaScript

The overarching goal of this work is to provide practical dynamic enforcement of secure information flow. To evaluate the approach, we have implemented an information-flow aware interpreter, *JSFlow*, for the full non-strict ECMA-262(v.5) [26], including information-flow models for the standard API. JSFlow is available online [41]. We have opted not to support strict mode, although it may simplify information-flow tracking. The reason is that its adoption is rather limited, and that it does not introduce obstacles for information-flow security. In addition, it is possible to run strict code using non-strict semantics.

JSFlow is itself implemented in JavaScript. The choice of language allows for flexibility in the deployment. We have explored the possibility of deploying the interpreter via browser extension, via proxy, via suffix proxy, and as a security library [47]. It is also possible to use *JSFlow* on the server side by running on top of, e.g., *node.js* [40]. The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [52].

The evaluation, detailed in Section 8, was performed using an experimental Firefox extension, *Snowfox*. To allow for experiments on actual web pages, the extension is accompanied with extensive stateful information-flow models for the APIs present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. The models of the standard and browser API are discussed in more detail in Section 7.

In addition to being used to enforce secure information flow on the client side, our implementation can be used by developers as a security testing tool, e.g., during the integration of third-party libraries. This can provide developers with detailed analysis of information flows on custom fine-grained policies, beyond the analysis from empirical studies [70,39,32,30] on simple policies.

The implementation is intended to investigate the suitability of dynamic information-flow control, and lay the ground for a full scale extension of the JavaScript runtime in browsers. A high-performance monitor would ideally be integrated in an existing JavaScript implementation like V8 or SpiderMonkey, but they are fast moving targets, focused on advanced performance optimizations. Instead, we believe that our JavaScript implementation finds a sweet spot between implementation effort and usability for research purposes. Although performance optimization is a non-goal in the scope of the current work, it is a worthwhile direction for future work.

To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript, together with stateful information-flow models for its standard execution environment.

Labels The simple two-level lattice of Section 4 does not suffice for actual web pages. Instead, the implementation uses a powerset lattice of information *origins*. A baseline policy is to consider all information to be public unless it originates from the user, e.g., information read from an input property, in which case it is labeled *user*. The default labeling can be overridden by explicit annotations in the HTML document.

The pc *handling* It is worthwhile to point out an interesting generalization in the *pc* handling in the implementation, when compared to the semantics presented in Section 4.

Every time the interpreter internally branches on labeled values, the control flow of the interpreter risks giving rise to implicit information flow that might be visible to the interpreted program. In order to tackle this, we replace the notion of the *program pc* with a *pc stack*. Whenever the *interpreter* branches on a security labeled value, its label is pushed onto the *pc* stack, where it remains until execution reaches a join point in the interpreter. Any side effects are governed by the join of all labels on the *pc* stack. Note that the novelty of this approach is not in the use of a stack for storing the *pc* (e.g., [43]), but the fact that it is the interpreter-internal information flow that is pushed onto the stack. Since all explicit and implicit information flows in the interpreted program are induced by explicit or implicit information flows in the interpreter, this generalizes and subsumes the standard notion of a program *pc*. For example, just as the type of a value decides which conversion methods to call, so does the value of the guard of a conditional statement decide which subprogram to execute. In both cases, the label of the value is pushed. See the implementation of `ToString` in Section 7.2 for an example of how the *pc* stack is used in the implementation.

7. The runtime environment: the standard API and the browser API

The standard JavaScript runtime environment mandates a number of built-in ECMAScript objects: Object, Function, Array, String, Boolean, Number, Date, RegExp, Error, and JSON. These objects are

accessible to a JavaScript program via identically named properties on the global object. Most of these objects have dual purposes by serving both as constructors and libraries containing various functions. In addition, the prototype property of the constructor objects provides the prototype for the constructed object. Like any standard compliant JavaScript interpreter, JSFlow provides a fully standard compliant execution environment via the global object. Being the runtime environment of an information-flow aware interpreter, it is necessary to track information flow into it. This entails being able to track information flowing into any of the objects provided or constructed from the provided (function) objects or functions, as well as tracking the information flow of the provided functions and methods.

While we could reimplement all of JavaScript's built-ins, it is more reasonable to defer as much work as possible to the underlying JavaScript engine, in which the interpreter itself runs. For some functionality, calling the corresponding functionality of the interpreter running JSFlow is necessary. This is true for functionality with side effects that cannot be modeled in JavaScript, such as interaction with the operating system, in the case of node.js, or the browser. For other functionality, it might not be possible, or overly approximative, from an information-flow perspective, to call the corresponding underlying functions.

For the command line version of JSFlow, the situation is depicted in Figure 12.

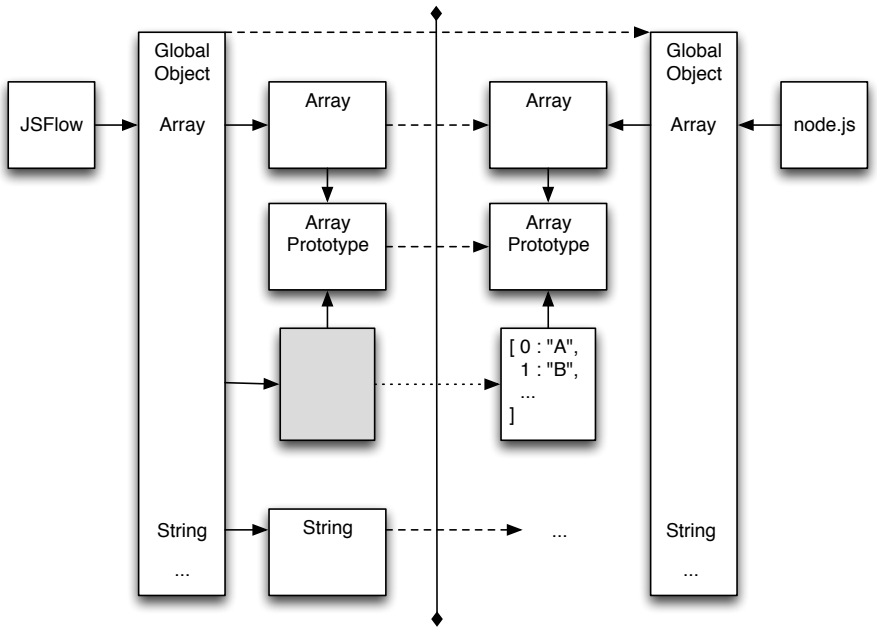

Fig. 12. The JSFlow runtime

The figure illustrates the relation between the JSFlow runtime environment and that of the underlying engine for a subset of the environment. The dashed lines represent the fact that parts of the functionality is deferred. In addition, the light gray rectangle represents a constructed array object. The dotted line shows that the internal value model of a JSFlow array object is a native array object. The solid line reflects that the internal prototype of the JSFlow array object is the Array prototype.

We refer to the implementation of the standard runtime environment (and any extension to it) as an information-flow aware model of the same. Such models must be created for any kind of library not written in JavaScript to connect the functionality of the library to the information-flow tracking. The

problem of modeling information flow in libraries is largely unexplored. Statically, libraries are typically handled by giving some form of boundary types to the interface of the library [54]. The precision and permissiveness of the enforcement of information-flow policies then depends on the expressive power of such boundary types. In this work, we have instead developed dynamic models that make use of actual runtime values to increase their precision.

7.1. Information-flow library models

In designing an information-flow model for a library, its precision must be balanced with its complexity and usability. Increased precision allows more permissive enforcement but typically adds to the complexity of using the library. For example, the user may need to supply security annotations, or otherwise be aware of the model itself. This results in a system that is harder to use and understand. On the other hand, being too imprecise risks having the enforcement reject too many secure programs, thus losing permissiveness. This also places a burden on the programmer, as she will have to work around the false positives.

A library model contains two parts: 1) a information-flow model of any internal state of the library, and 2) information-flow models for all functions of the library. Based on the discussion above, we distinguish between function models, depending on the extent the standard functionality can be deferred to the corresponding functionality in the underlying library. In particular, we categorize information-flow models for library functions into two different types: *shallow* models and *deep* models. Shallow models describe the operations on labels and label state in terms of the boundary values and types of the parameters to the library function, whereas deep models may compute internal, intermediate values such as private attributes of objects or local variables inside library code. The model may perform or replicate a part of the computation done by the library, in order to obtain a more precise model.

The remainder of this section discusses key insights gained from modeling the built-in JavaScript API, as well as browser APIs, and gives examples where deep models are necessary to yield useful precision.

7.2. JavaScript standard API

In addition to the core language, JSFlow implements the API mandated by the standard. The information-flow models of the standard API range from simple shallow models to more advanced deep models. Below, String and Array illustrate shallow and deep models, respectively.

String object The String object acts as a wrapper around a primitive string providing a number of useful operations, including accessing a character by index. We model the internal label state of String objects with a single label matching the security model of its primitive string.

Several of the methods of String objects are shallow models: after converting the parameters, they are passed to the corresponding native method. For instance, consider `toLowerCase` that converts a string to lowercase. While we could iterate through the entire string converting character by character, we may as well call the corresponding function.

```
1 function toLowerCase(thisArg,args) {
2   conversion.CheckObjectCoercible(thisArg);
3   var S = conversion.ToString(thisArg);
4   var L = S.value.toLowerCase();
5   return new Value(L,S.label);
6 };
```

This function model is an example of a shallow model. It first performs all necessary conversions, calls the underlying `toLowerCase` and returns the result labeled. Note that the model cannot defer the conversion to the underlying model, since the conversion may trigger potentially side-effectful callbacks that must be invoked in the right security context.

As another example of an essentially shallow model, consider the `slice` method. The slice methods takes two indices and returns the slice of the string between them. We highlight the implementation of `slice` by means of examples.

First, consider the following program, which passes an object to `slice` whose `valueOf` method will return a secret.

```
ix = { valueOf : function() { return h; }};
var l = '0123456789'.slice(ix,ix+1);
```

This example tries to exploit the conversion to numbers that `slice` performs on its arguments, which invokes `valueOf` when they are objects. In order to model this flow properly, the security label of the value returned by `valueOf` must be taken into account in the result of `slice`. In the example above, `slice` would return a secret value.

In addition, it is important that the security context of the calls made by `slice` include the labels of the indices. Otherwise, side effects in `valueOf` may leak. In the following example, assume that `ix` is secret, and that it is chosen to be either an object or a number depending on h.

```
var ix; var l = false;
if (h) {
  ix = { valueOf :
         function() { l = true; return 0; } }
} else { ix = 0; }
'0123456789'.slice(ix,ix+1);
```

In the implementation of `slice` below, lines 7 to 19 perform the mandated conversions. Note how the conversion of `end` on line 17 is performed in the context of `end.label`, since the value of `end` controls whether the conversion is performed or not. Omitting this opens up the coercion for exploits. Once all conversions have been completed, the actual functionality is deferred to the underlying library on line 21, and the result is returned appropriately labeled.

```
1  function slice(thisArg,args) {
2    var c = monitor.context;
3
4    var start = args[0] || new Value(undefined,bot);
5    var end = args[1] || new Value(undefined,bot);
6
7    conversion.CheckObjectCoercible(thisArg);
8    var S = conversion.ToString(thisArg);
9    var len = S.value.length;
10
11   var intStart = conversion.ToInteger(start);
12
13   c.pushPC(end.label);
```

```
14      if (end.value === undefined) {
15        end = new Value(len, lub(S.label, end.label));
16      } else {
17        end = conversion.ToInteger(end);
18      }
19    c.popPC();
20
21    var str = S.value.slice(start.value, end.value);
22    var lbl = lub(S.label,start.label,end.label);
23    return new Value(str,lbl);
24  };
```

Let us return to side effects in the conversion of the indexes performed on line 11 and conditionally on line 17. Here, the security context of the call to `valueOf` when converting `ix` must reflect the label of `ix`. We ensure this in the internal functions `ToString` and `ToNumber` (called by `ToInteger`), used by `slice`. For example, the actual security context increase occurs in `ToNumber` on line 6, where the argument label is pushed onto the *pc* stack:

```
1  function ToNumber(x) {
2    if (typeof x.value !== 'object') {
3      return new Value(Number(x.value),x.label);
4    }
5
6    monitor.context.pushPC(x.label);
7      var primValue = ToPrimitive(x, 'number');
8    monitor.context.popPC();
9
10   return new Value(Number(primValue.value), primValue.label);
11 }
```

This ensures that `valueOf` in the example above will be called in the right security context.

Array Array objects are list-like objects that map numerical indices to values. Arrays have a special link between the `length` property and the mapped values: writing an element past the length of the array will increase the length property accordingly, and decreasing the length property will remove elements from the end of the array. We model the internal state of the array as an ordinary object, while catering for the connection between the `length` property and the indices. Arrays are mutable and equipped with methods for performing different operations on the elements of the array in different orders. This allows for complicated interplays with accessor properties, which calls for the use of deep models. Consider, for instance, the following example:

```
x = [h]; l = false;
Object.defineProperty(x,1,
  { get : function() { l = true; return 0}});
x.every(function (x) { return x; });
```

The `every` method of arrays invokes a function on each element, until either the list is exhausted or it returns a value convertible to *false*. Since all values are convertible to one of *true* or *false*, knowing that

an element with index greater than 0 is read reveals that the function returned a *true*-convertible value for all lower indices. By populating an array with a secret followed by a getter, the true value of the secret could be observed. In the example, the first element contains the secret boolean `h` and the second element is a getter that sets `l` to *true*. Since the getter is only invoked in case h is *true*, this effectively copies h to l. For this reason, a shallow model cannot be used. Each successive call of the iterator function must be called in the accumulated context of the previous results, which is not possible, if we simply delegate the computation to the primitive `every` method. Instead, we must use a deep model, illustrated below with an excerpt of the inner loop of `every`.

```
1  function every(thisArg,args) {
2
3    ...
4
5    while (k.value < len.value) {
6
7      ...
8
9      if (kPresent.value) {
10       var kValue     = O.Get(k);
11       var testResult = fn.Call(_this, [kValue, k, O]);
12       var b = conversion.ToBoolean(testResult);
13       monitor.context.labels.pc.lubWith(b.label);
14       label.lubWith(b.label);
15
16       if (!b.value) {
17         monitor.context.popPC();
18         return new Value(false,label);
19       }
20     }
21     k.value++;
22   }
23
24   c.popPC();
25   return new Value(true,label);
26 }
```

Notice how line 12 converts the result of the function to a boolean, how line 13 uses the label of the result to accumulatively increase the top of the *pc* stack, and how line 14 accumulates the label used for the returned value at line 18. Since `ToBoolean` may have side effects, the accumulation is important. It cannot be implemented if the functionality is deferred to the corresponding underlying function.

7.3. Browser APIs

The execution environment provided by browsers is an extension of the built-in JavaScript environment. To a certain extent, what is provided is browser specific, while some parts are standardized by the World Wide Web Consortium (W3C). Although many of the extensions are fairly straightforward to model from an information-flow perspective, offering similar challenges as the standard library, a notable

exception is the implementation of the Document Object Model (DOM) API [38]. The DOM is a standard describing how to represent and interact with HTML documents as objects. The DOM is a central data structure to all web applications. A large part of a web application typically deals with shuttling data to and from the DOM and responding to events generated by the DOM, as the user interacts with it. Tracking information flows to and from the DOM is thus vital to having information-flow tracking that is useful for real web applications.

Sources and sinks in the DOM Being an object model of the HTML document that makes up the graphical interface of the web application, the DOM naturally contains many sources of potential sensitive information. In particular, essentially all forms of user provided input will be represented in the HTML document. To exemplify, consider the user name and password fields used to log into the web application, or a form used to collect payment information for a purchase. What parts of the HTML document that should be considered sensitive is application dependent. JSFlow offers the possibility to label individual DOM elements, which allows for fine-grained application specific policies.

In addition to information sources, DOM elements may act as information sinks. A natural example of such sinks are forms that submit the entered information to a given URL. While forms are examples of explicit sinks, there are also implicit sinks. Any element that fetches remote resources can be used as an implicit sink by encoding the information in the URL of the resource. The most prominent example of use of implicit sinks are to encode information in the URL of image elements and use it to fetch a zero pixel large image. Among other, this technique is used by many analytics services in order to circumvent the same-origin policy.

However, sources and sinks are not exclusive to the DOM API, but can can be found in other browser APIs as well. For instance, an API that gives access to information about the specifics of the execution environment, such as the Navigator object, can be used for fingerprinting and vulnerability scanning. Under certain circumstances such APIs may be considered sensitive sources. Similarly, APIs like XMLHttpRequest and Web Sockets provide general explicit sinks.

Non-local models: live collections In addition to acting as a data structure, the DOM also provides a rich set of behaviors. In particular, several features of the DOM force information-flow models to be *non-local*, i.e., operations on a certain element in the tree may require updates to the model of other elements in the tree. A prime example of such a feature is *live collections*.

The DOM standard [38] specifies a number of methods for querying the DOM for a collection of certain elements. Collections are represented as objects that behave much like arrays, with one big exception: as the DOM is modified, collections are updated to reflect the current state of the DOM.

For example, `getElementsByName` returns a live collection of elements with a particular name attribute. If a script changes the name attribute of an element in the page, the corresponding live collections automatically reflect the change. To appreciate this, consider the following example based on a web page containing a *div* element with id and name 'A'.

```
<div id='A' name='A'></div>
```

When the following code is executed in the context of this page, it encodes the value of h in the length of the live collection returned by `getElementsByName`.

```
c = document.getElementsByName('A');
if (h) { document.getElementById('A').name = 'B'; }
```

This is achieved by conditionally changing the name of the *div* from 'A' to 'B'. Initially, the collection stored in c has length 1, since there is one element in the document named 'A'. After the name change, however, the collection contains no elements. Importantly, this is done without any direct interaction with the collection itself; only the *div* element is referenced and modified.

The security model of live collections must interact with the model for the DOM, making it non-local. We keep in each DOM node a map from queries generating live collections, such as `getElementsByName('A')`, to the label representing how that node's subtree affects that query.

Going back to the above example, the document (the root node of the DOM tree) maintains a map from names to labels. If this map associates 'A' with public, the interpreter will stop execution on the attempted name change, since it would be observable on existing public live collections. On the other hand, if this map associates 'A' with secret, the name change is allowed. In this case, however, any live collection affected is already considered secret.

Live collections are just one example of non-local behavior in the DOM. For another example, several DOM elements expose properties that are actually computed from state stored elsewhere in the DOM. For instance, a form element exposes values of nested input properties as properties on the form element itself. Also, some element attributes, which are DOM nodes of their own, are exposed as properties on the containing element. The security model of DOM nodes must properly model these cases and label the result appropriately. If we blindly accessed the properties in the underlying API, we could return secret values stripped of their security label.

Beyond handwritten models At the moment, the library models are handwritten based on the description of the API functions. There are two main challenges related to handwritten library models.

First, without some form of automation the undertaking of producing a library model for a large API is very labor intensive. Already supporting the full standard API of JavaScript is a relatively large effort. Around 30% of the JSFlow code base[1] consists of library models of the standard API. Full browser API support or support for JSFlow on the server side would significantly increase this number. In addition, a fair portion of the library code is boilerplate code to adhere to the object and function models of JSFlow.

Second, handmade library models are based on analysis and assumptions on the functionality of the library. For shallow and non-local models in particular this may be an issue. Any mismatches between the model and the functionality may open up possibilities for attacks. To avoid this, a more formal connection between the model and the library would be beneficial.

With respect to the first challenge, we envision automatic generation of library models from higher level descriptions. Such descriptions could range from pure functions on the involved security labels, supporting only the most basic shallow models, to functions that take values and side effects into account and even a fully fledged programming language. The design of the model language is a trade-off between expressiveness and benefit. For instance, to support fully deep models, the latter may be necessary, but then the gain over fully handwritten models may be relatively small.

With respect to the second challenge, for cases where the source code or some other functional description of the library is available, we envision using static analysis, e.g., abstract interpretation, to automatically generate library models that are guaranteed to capture the information flows of the library. When possible, this approach would also address the first challenge. We see both approaches as interesting but challenging future directions.

[1] JSFlow code base is around 12 thousand lines of JavaScript

8. Evaluation

This section starts by reporting on two types of experiments, performed in September 2013, one to explore different policies for user data and the other to govern user tracking by third-party scripts. We then go on to discuss general security considerations, trade-offs for dynamic enforcement, and going beyond dynamic analysis.

For the case studies, we have created *Snowfox*, a Firefox extension that uses JSFlow as the execution engine for web pages. Snowfox is based on Zaphod [53] and turns off Firefox's native JavaScript engine. Instead, the extension traverses the page as it loads and executes scripts using JSFlow.

This provides a proof-of-concept implementation that allows us to study the suitability of dynamic information flow on actual JavaScript code. When an information-flow policy is violated, the extension can respond in various ways, such as simply logging the leak, silently blocking offensive HTTP requests or stopping script execution altogether.

Performance When deploying runtime analyses in practice, the execution overhead brought by the analysis is important. While the focus of this paper is to develop and investigate the limits of dynamic information-flow control in a realistic setting, making the analysis practically useful is an important part of the long term goal.

Once we have established that dynamic information flow is possible from a permissiveness perspective, it remains to make it practically useful. For our current purposes, we have found the speed of JSFlow adequate. In particular, it did not hinder us in manually interacting with web pages.

Compared to a fully JITed JavaScript engine, JSFlow is slower by two orders of magnitude on the tested pages. The comparison is, however, not illustrative of the cost of dynamic information-flow tracking. Even without any tracking JSFlow, being an unoptimized interpreter written in JavaScript, would be outperformed by a highly optimized JIT compiler. Rather, in order to perform a more meaningful comparison, the implementation needs to be evaluated against a comparable baseline. There are two paths forward. Either JSFlow is compared against a version of JSFlow, where the information-flow tracking is turned off or one of the existing JavaScript engines are extended to support the information-flow tracking performed by JSFlow. In this way, a more reasonable estimation of the relative cost of information-flow tracking can be reached.

User input processing We have evaluated the interpreter on several web applications that calculate loan payments, given input provided by the user. Such applications do not rely on external data. Running the interpreter under different policies reveals some security-relevant differences between applications and demonstrates our interpreter's ability to enforce them on real JavaScript code. As a baseline policy, we use a stricter version of SOP, where communication via requests such as creating image and script tags is not allowed if it involves information derived from user inputs.

We have found three main classes of loan calculators: (i) Scripts that do all calculations in the browser: no data is submitted anywhere. (ii) Scripts that submit user data to the original host for processing, but not to third parties. (iii) Scripts that submit data to a third party, e.g., for collecting statistics, or allow third-party scripts to access user data. At the time of writing, example web pages for each class are (i) `http://www.halifax.co.uk/loans/loan-calculator/` and `http://www.asksasha.com/loan-interest-calculator.html`; (ii) `http://www.tdcanadatrust.com/loanpaymentcalc.form`; and (iii) `http://mlcalc.com/`. A calculator of the first class works under a policy that allows no data to leave the browser. On the other hand, the second class needs to send data to its origin server, but still works under the strict SOP policy. The third class requires a more liberal policy.

Web pages commonly use Google Analytics to analyze traffic. To do so, the web page loads a script provided by that service. This script triggers an image request conveying tracking information to Google. As long as no data about user input is contained in the request, scripts of type (i) and (ii) can still use Google Analytics under our interpreter. A calculator in class (iii) that tries to log user inputs, or any derived values, to Google Analytics is prevented from doing so by our interpreter. Flows are correctly tracked inside the Google Analytics script, and the interpreter does not allow creating an image element with such data in the source URL.

One of the sites in our tests, `mlcalc.com`, did send user inputs to Google Analytics, but indirectly. The user inputs were first submitted to `mlcalc.com`, but the following page included JavaScript code that logged them to Google Analytics. The flow in this case was essentially server-side, so some server-side support is needed to track them. Our monitor supports upgrade annotations, i.e. a server can explicitly label some of its data as being derived from sensitive user inputs. We note that JSFlow can also be run on the server side, e.g. via *node.js*, for an end-to-end solution.

When the host is not trusted for user data, a still stricter policy can be utilized, namely that no user data should leave the browser. Under this policy, the interpreter correctly stops even the submission of a form. This still allows calculators of type (i), which compute everything client-side.

Behavior tracking via JavaScript Users are often unaware of information sent from their browser, e.g., for tracking purposes. As an example, the service Tynt offers a script to inject links back to the including website, into content copied to the system clipboard. This service is used on popular web sites such as the Financial Times (`www.ft.com`). As the clipboard API in JavaScript does not provide append functionality, this script relies on having read access to the copied data, constituting a source of information. However, transparent to the user, the script also creates a request via an image to `tynt.com`, constituting an information sink, where it logs the copied data, together with a tracking cookie unique to the user across different websites using Tynt.

Our implementation supports all the necessary browser APIs used by Tynt's script and is able to detect this behavior. Since the selection is chosen by the user, the data copied is considered secret. When the script attempts to communicate this data back to Tynt, the interpreter detects the leak and throws a security error.

Security considerations At the core of JSFlow is the formal model of information-flow tracking for a language with records and exceptions described in Section 4. The formal model includes a dynamic type system for a core of JavaScript that has been proved sound. As discussed in Section 6, much of the extension to full JavaScript is via sound primitive constructions, while extensions not expressed in terms of sound primitive constructions are built using a small number of core principles.

For now, the correctness of JSFlow is only verified using testing. This is true both for the functional correctness, as well as for the soundness of the information flow. However, the connection between the interpreter-internal information flow and the information flow of the interpreted language provides an indication of a potential way of verifying the implementation of JSFlow using a specialized static type system. The basic idea is that any explicit or implicit flow in the interpreted language is manifested as an explicit or implicit flow in the interpreter. By connecting the representation of labeled values to a type system tailor-made for checking JSFlow, it will be possible to make sure that all such flows are properly taken into account.

Trade-offs for dynamic enforcement A key question is whether it is possible to implement an interpreter with enough precision for the enforcement to be usable and permissive. Our interpreter indicates that tracking flows in real-world applications is feasible. Being a flexible language, JavaScript provides many

implicit ways for information to flow, in particular when combined with the rich APIs of the browser environment. Our interpreter successfully addresses such features, while being reasonably precise to allow real scripts to maintain their utility.

However, dynamic enforcement comes at a price of inherent limitations. In particular, there are limits on sound and precise propagation of labels under secret control [60], leading to a common restriction of enforcement known as no sensitive upgrade (NSU) [76,3]. We have found that legacy scripts sometimes encounter such situations, even if they do not always leak confidential data. In such scripts, upgrade statements must be injected [13].

For the experiments, we have developed a simple hybrid analysis, inspired by [43], that upgrades the security labels of local variables before entering secret contexts. Despite the simplicity of the approach, it shows good promise and reduces the number of false positives significantly. Driven by the positive initial experiment, we have since developed a hybrid dynamic approach that is able to handle all remaining false positives we encountered in our experiments. We refer the reader to [34] for a detailed explanation of the upgrades we have identified, and how they can be handled. A full scale implementation and evaluation of a hybridized JSFlow is underway [41].

Without hybridization, we estimate that the majority of pages need annotations. Our initial results show that the hybrid approach is able to remove a significant number of false positives. This said, due to the undecidability of the problem, the hybrid approach cannot remove all false positives. Regardless, a small number of remaining false positives is not necessarily a deal breaker. Responsible site owners may inject upgrade instructions into their code in order to be able to offer their users the added protection of information-flow control.

For the sake of the experiment, we have decorated the scripts with upgrade instructions via proxy. Most of the annotations upgrade non-variable locations, such as object properties. However, some of the annotations concern dangerous information flow on the server side. Such information flows cannot be handled by client side information-flow control in isolation, but additionally requires the server to track information flow. In principle, JSFlow can be used to provide client-server end-to-end information-flow security. By leveraging that JSFlow is implemented in JavaScript, a server-side JSFlow runtime environment can be created. For example, the popular *Node.js* [40] JavaScript runtime is a natural fit. This allows the server to tie into the information-flow tracking on the client side, providing a complete client-server solution.

We have also discovered two cases where the interpreter detected flows that originate from a nullity check on user input. Since these checks were performed relatively early in the execution, they resulted in much derived data to be labeled as secret. However, in both cases, we had found that the checks did not leak, since no action of the user could cause the checked values to be null. The checks were only safeguards to prevent failures due to browser differences or bugs in the scripts. We had thus added declassification annotations to allow these benign flows. Such cases are easily handled by value-sensitivity [8] over value types. In the future, we aim to enrich JSFlow with this feature. In combination with the mentioned hybrid aspect, a value-sensitive JSFlow pursues a more practical and permissive information-flow control enforcement.

9. Related Work

Amongst a large body of research on language-based approach to information-flow security [63], 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 [60] 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.

Fenton [28] discusses purely dynamic monitoring for information flow but does not prove noninterference. Volpano [71] considers a purely dynamic monitor to prevent explicit (but not implicit) flows. In a flow-insensitive setting, Sabelfeld and Russo [64] 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 [2], as well as timeout instructions [59].

In previous work, Russo et al. [61] 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 [3,4] 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. Bichhawat et al. [10] explore generalizations of permissive upgrade for the case of a multi-level security lattice. Austin and Flanagan [4] 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. [48] show how to inline a no-sensitive upgrade monitor into programs in a language with dynamic code evaluation. Their approach is based on the use of *shadow variables* to keep track of the security labels.

Along similar lines, Chudnov and Naumann [17] present a method for inlining no-sensitive upgrade monitors into programs written in ECMA-262 (v.5) [26] and (selected parts of the) API. The approach is based on *boxing*, which they argue is beneficial due to the way modern JIT compilation works.

Bohannon et al. [15] present a flow-insensitive static analysis for JavaScript-like event systems. Rafnsson and Sabelfeld [57] 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 formal modeling of the semantics of JavaScript.

On one hand, Maffeis et al. [45] give the first detailed semantics for full JavaScript. It is a full account of the ECMA-262 (v.3) standard [25], which faithfully models the, sometimes slightly unusual, behavior of JavaScript programs. Similar in spirit is the work by Bodin et al. [14] that formalizes the full ECMA-262 (v.5) standard [26] in Coq. In addition, this formalization allows for the extraction of a standard compliant reference implementation.

On the other hand, Guha et al. [33] 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. [75] also formulate a semantics in terms of a lambda calculus. Contrary to Guha et al. [33], 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. While there is much work on safe sublanguages, e.g, Caja [51], ADSafe [20], Gatekeeper [31], and work on refined access control for JavaScript, as in, e.g., ConScript [50] and JSand [1], we recall our motivation from Section 1 for the need of information-flow control beyond access control. In the rest, we focus on information-flow tracking for JavaScript. Hence, our work is not a safe sublanguage approach. Further, we chose to be faithful to the ECMA-262 standard (v.5).

Practical JavaScript analysis 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. [70] modify the source code of the Firefox browser to implement a 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.

Mozilla's ongoing project FlowSafe [27] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [3] at its core. Our coverage of JavaScript and its APIs provides a base for fulfilling the promise of FlowSafe in practice.

Chugh et al. [18] 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.

Yip et al. [74] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server.

Mash-IF, by Li et al. [44], is an information-flow tracker for client-side mashups. With policies defined in terms of DOM objects, the enforcement mechanism is a static analysis for a subset of JavaScript and treats as blackboxes the language constructs outside this subset. Executions are monitored by a reference monitor that allows deriving declassification rules from detected information flows. An advantage of this approach is fine-grained control at the level of individual DOM objects. At the same time, the imprecision of the static analysis leads to both false positives and negatives, opening up for attackers to bypass the security mechanism.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [24] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. The goal is to prevent malicious extensions from breaking confidentiality. Bandhakavi, et al. [6] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [39] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [18], the analysis is based on code rewriting that inlines checks for data produced from sensitive sources not 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.

Guarnieri et al. [32] present Actarus, a static taint analysis for JavaScript. An empirical study with around 10,000 pages from popular web sites exposes vulnerabilities related to injection, cross-site scripting, and unvalidated redirects and forwards. Taint analysis focuses on explicit flows, leaving implicit flows out of scope.

Just et al. [42] develop a hybrid analysis for a subset of JavaScript. A combination of dynamic tracking and intra-procedural static analysis allows capturing both explicit and implicit flows. However, the static analysis in this work does not treat implicit flows due to exceptions.

Le Guernic et al. [43] present a hybrid information-flow analysis for a language without a heap. A static analysis is employed to identify write targets before entering elevated contexts. Due to the simplicity of the language, the approach is able to syntactically detect all possible write targets.

Hedin et al. [34] present a hybrid information-flow analysis for a core of JavaScript. Similar to Le Guernic et al. [43], the analysis employs a static component to identify write targets before entering elevated contexts. However, due to the complexity of the language, it is not always possible to approximate all write targets. Instead, the approach relies on a base-line NSU monitor for soundness, allowing more freedom in the static component.

Another hybrid analysis for JavaScript is developed by Besson et al. [9]. With the goal of limiting the amount of web browser fingerprinting information leakage, their system tracks the amount of secrecy in each variable, instead of a single label, using Quantitative Information Flow. As in [43] and [34] the static part of the enforcement is triggered at runtime, when the context is elevated, to improve the precision of the dynamic analysis.

Pushing the boundary of permissiveness, Bello et al. [8] develop and explore the notions of value sensitivity and observable abstract values, showing how to systematically apply these notions to improve the permissiveness of hybrid and dynamic analyses.

Bichhawat et al. [11] present an information-flow analysis for JavaScript bytecode. The analysis is implemented as instrumented runtime system for the WebKit JavaScript engine. The implementation includes a treatment of the permissive-upgrade check.

The empirical studies [70,39,32,30] provide clear evidence that privacy and security attacks in JavaScript code are a real threat. As mentioned earlier, the focus is the *breadth*: trying to analyze thousands of pages against simple policies. Complementary to this, our goal is in-*depth* studies of information flow in critical third-party code (which, like Google Analytics, might be well used by a large number of pages). Hence, the focus on dynamic enforcement, based on a sound core [37], and the careful approach in fine-tuning the security policies to match application-specific security goals.

We believe that the road to bridging the gap between formal and empirical approaches is dynamic information-flow tracking, possibly with hybrid helper components. For a language like JavaScript, purely static analysis is hardly feasible, as argued by this paper and others [67], while dynamic analyses provide possibilities for precisely recording relations between data in a given trace.

Secure multi-execution In an orthogonal yet promising effort, Devriese and Piessens [23] 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. [12] implement secure multi-execution for the Featherweight Firefox [16] model. Austin and Flanagan [5]

propose faceted values to accommodate the benefits of 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 formalize the approach 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.

De Groef et al. [30] present *FlowFox*, a Firefox extension based on secure multi-execution and perform practical evaluation of user experience when simpler policies (such as labeling the cookie as sensitive) are enforced.

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. Recent efforts are focused on generalizing secure multi-execution and introducing possibilities of declassification [58,69].

Web tracking Web tracking is subject to much debate, involving both policy and technology aspects. We refer to Mayer and Mitchell [49] for the state of the art. In this space, the Do Not Track initiative, currently being standardized by World Wide Web Consortium (W3C), is worth pointing out. Supported by most modern browsers, Do Not Track is implemented as an HTTP header that signals to the server that the user prefers not to be tracked. However, there are no guarantees that the server (or third-party code it includes) honors the preference.

Origins of the work and its immediate successors As mentioned before, this work merges, expands and improves our previous work on information-flow tracking for a core of JavaScript [37] and subsequent implementation [36]. The added value of this paper is spelled out in Section 1. Recently, we have explored different architectures for inlining security monitors in web applications [47]. The architectures allow deploying any monitors, implemented in JavaScript in the form of security-enhanced JavaScript interpreters, under different architectures. We have investigated the pros and cons of deployment as a browser extension, as a proxy, as a service, and an integrator-driven deployment. The pros and cons reveal security trade-offs and usability trade-offs. We have shown how to instantiate the general deployment approach with the JSFlow monitor.

Thiemann has recently explored program specialization techniques to improve JSFlow's performance [68], yielding a speedup between the factor of 1.1 and 1.8.

10. Conclusions and future work

In line with our overarching goal and individual objectives, we have explored and connected the theory and practice of information-flow security for JavaScript.

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.

We have presented JSFlow, a security-enhanced JavaScript interpreter, written in JavaScript. To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

We have demonstrated that JSFlow enables in-depth understanding of information flow in practical JavaScript, including third-party code such as Google Analytics, jQuery, and Tynt.

Future work points to two particularly interesting directions. First, our case studies with third-party scripts provide valuable insights into possibilities and limitations of dynamic information-flow enforcement. It makes it clear that hybrid (dynamic/static) information-flow tracking is a promising direction, enabling dynamic monitors to increase permissiveness by helper static analysis.

We have showed how to model the libraries, as provided by browser APIs, by a combination of shallow and deep modeling. Our findings lead to possibilities of generalization: future work will pursue automatic generation of library models from abstract functional specifications.

Acknowledgments This work was funded by the European Community under the ProSecuToR and WebSand projects, and the Swedish agencies SSF and VR.

References

[1] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In R. H. Zakon, editor, *ACSAC*, pages 1–10. ACM, 2012.

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

[3] 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.

[4] 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.

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

[6] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9):91–99, 2011.

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

[8] L. Bello, D. Hedin, and A. Sabelfeld. Value sensitivity and observable abstract values for information flow control. In *Proc. of the International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, Nov. 2015.

[9] F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 240–254, June 2013.

[10] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, July 2014.

[11] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *POST*, pages 159–178, 2014.

[12] 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.

[13] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In S. Foresti, M. Yung, and F. Martinelli, editors, *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2012.

[14] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised javascript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

[15] 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.

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

[17] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 2105.

[18] 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.

[19] 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.
[20] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.
[21] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
[22] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
[23] D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. IEEE Symp. on Security and Privacy*, May 2010.
[24] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391. IEEE Computer Society, 2009.
[25] ECMA International. ECMAScript Language Specification, 1999. Version 3.
[26] ECMA International. ECMAScript Language Specification, 2009. Version 5.
[27] B. Eich. Flowsafe: Information flow security for the browser. `https://wiki.mozilla.org/FlowSafe`, Oct. 2009.
[28] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.
[29] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.
[30] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM CCS*, 2012.
[31] 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.
[32] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In M. B. Dwyer and F. Tip, editors, *ISSTA*, pages 177–187. ACM, 2011.
[33] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, June 2010.
[34] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *Proc. IEEE CSF*, July 2015.
[35] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs (full version), 2016. `http://www.cse.chalmers.se/~andrei/jsflow-jcs16.pdf`.
[36] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. 29th ACM Symposium on Applied Computing*, 2014.
[37] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF*, pages 3–18, June 2012.
[38] A. L. Hors and P. L. Hegaret. Document Object Model Level 3 Core Specification. Technical report, The World Wide Web Consortium, 2004.
[39] 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.
[40] Joyent, Inc. Node.js. `http://nodejs.org/`.
[41] The JSFlow project. Located at `http://www.jsflow.net/`.
[42] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information Flow Analysis for JavaScript. In *Proc. ACM PLASTIC*, pages 9–18, USA, 2011. ACM.
[43] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.
[44] Z. Li, K. Zhang, and X. Wang. Mash-IF: Practical information-flow control within client-side mashups. In *DSN*, pages 251–260, 2010.
[45] 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.
[46] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.
[47] J. Magazinius, D. Hedin, and A. Sabelfeld. Architectures for inlining security monitors in web applications. In *ESSoS*, Lecture Notes in Computer Science. Springer, 2014.
[48] 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.
[49] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE SP*, pages 413–427. IEEE Computer Society, 2012.
[50] L. A. Meyerovich and V. B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE SP*, pages 481–496. IEEE Computer Society, 2010.

[51] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, 2008.
[52] Mozilla Developer Network. SpiderMonkey – Running Automated JavaScript Tests. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests, 2011.
[53] Mozilla Labs. Zaphod add-on for the Firefox browser. http://mozillalabs.com/zaphod, 2011.
[54] 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.
[55] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *ACM CCS*, pages 736–747, Oct. 2012.
[56] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.
[57] 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.
[58] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *CSF*, pages 33–48, 2013.
[59] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.
[60] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.
[61] 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.
[62] 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.
[63] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
[64] 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.
[65] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, Sept. 1975.
[66] Taboola. Update: Taboola Security Breach - Identified and Fully Resolved. http://taboola.com/blog/update-taboola-security-breach-identified-and-fully-resolved-0, June 2014.
[67] 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.
[68] P. Thiemann. Towards specializing javascript programs. In *PSI*, LNCS. Springer-Verlag, 2014.
[69] M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *CSF*, 2014.
[70] 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.
[71] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.
[72] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
[73] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko, and B. Karp. Toward principled browser security. In *Proc.HotOS*, 2013.
[74] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys*, pages 233–246, USA, 2009. ACM.
[75] 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.
[76] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.