UNIVERSITY OF NICE - SOPHIA ANTIPOLIS
DOCTORAL SCHOOL STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

P H D T H E S I S

to obtain the title of

Doctor of Computer Science

of the University of Nice - Sophia Antipolis

Defended by

José FRAGOSO SANTOS

Toward Enforcing Secure Information Flow in Client-Side Web Applications

The Title In French Will Take At Least Two Lines

Advised by Tamara REZK, and Ana ALMEIDA MATOS

prepared at INRIA Sophia Antipolis, Team INDES

defended on December 8th, 2014

Jury :

President :	Name SURNAME	-	Title, Institute
Reviewers :	Name SURNAME	-	Title, Institute
	Name SURNAME	-	Title, Institute
Examiners :	Name SURNAME	-	Title, Institute
	Name SURNAME	-	Title, Institute
Advisors :	Tamara REZK	-	Title, Institute
	Ana ALMEIDA MATOS	-	Title, Institute
Invited :	Name SURNAME	-	Title, Institute
	Name SURNAME	-	Title, Institute

Abstract

We address the issue of enforcing confidentiality and integrity policies in the context of client-side web applications. Since most web applications are developed in JavaScript, we study static, dynamic, and hybrid enforcement mechanisms for securing information flow in a core of JavaScript that retains its defining features and that we call Core JavaScript. Specifically, we propose: **(1)** a monitored semantics for dynamically enforcing secure information flow in Core JavaScript as well as source-to-source transformation that inlines the proposed monitor, **(2)** a type system that statically checks whether or not a program abides by a given information flow policy, and **(3)** a hybrid type system that combines static and dynamic analyses in order to accept more programs than its fully static version.

Most JavaScript programs are designed to be executed in a browser in the context of a Web page. These programs often interact with the Web page in which they are included via a large number of external APIs provided by the browser. The execution of these APIs usually takes place outside the perimeter of the language. Hence, any realistic analysis of client-side JavaScript must take into account possible interactions with external APIs. We present a general methodology for extending security monitors to take into account the possible invocation of arbitrary APIs and we then apply this methodology to an important fragment of the DOM Core Level 1 API.

Acknowledgments

Here go the acknowledgments...

Contents

List of Figures

Chapter 1

Introduction

Contents

Web applications hold a prominent spot in the Internet of today. They are being increasingly used by people in their everyday lives to accomplish all sorts of tasks, including e-mailing, word processing, online banking and shopping, and many, many more. While some of these applications do not necessarily mandate a high level of security, there are those for which it is of paramount importance. Security of web applications is, therefore, an important and highly applicable research topic, and for us to be able to address it properly, we need to take a closer look at their general structure.

Most web applications are composed of several different programs, called scripts, that need not share the same origin. Some of these scripts can even be loaded from third-party code providers at runtime; this is the case, for example, when it comes to online advertisements. The code whose origin coincides with that of the web page is called the *integrator*, whereas each external script is called a *gadget*. Using gadgets in a web application is not mandatory, but if any are involved, it is then the job of the integrator to patch them all together in order to generate the web application, and such a web application is called a *web mashup*. The programming language typically used for the implementation of web mashups is JavaScript [5th edition of ECMA 262 June 2011 2011, 3rd edition of ECMA 262 1999] — a widely used programming language supported by all of the major browsers.

What can be said about the relationship between using gadgets in a web application and its security properties? The fact most pertinent to this question is that gadgets can be loaded at runtime and can even depend on the input given by the user. This is commonplace for instance, for online advertisements, loaded from ad servers that use various data mining techniques to determine which advertisements should be displayed to which user. Therefore, it is impossible for the developer of such web applications to know *a priori* which third-party code will be executed. This architectural style of modern web applications can raise serious security issues — malicious third-party programs can compromise the integrity and confidentiality of the user's resources. Illustratively, a recent study by Jang *et al* [Jang 2010] has shown that many websites, including some in the Alexa global top-100, exhibit privacy-violating security vulnerabilities.

In light of this security-critical situation, a shared interest exists between web application developers and users alike in the enforcement of isolation properties that guarantee that confidential resources are not leaked to untrusted parties and that high-integrity resources are not modified based on low-integrity data coming from untrusted gadgets. In fact, the central concept in the web application security model, the Same Origin Policy (SOP) [Barth 2011], was designed to provide precisely this type of guarantees. Roughly, this policy states that a script loaded from one origin is not allowed to access or modify resources obtained from another origin. Here, as in [Yang 2013], we refer to this definition as the *strict* SOP. While a full implementation of the strict SOP would definitely solve most of the security issues that wreak havoc on modern web applications, it would, unfortunately, also severely constrain one of their essential features, that being the interaction between scripts of different origins within a web page. As it is, in order to allow for cross-origin communication, the browser security model includes many exceptions to the strict SOP. For instance, current browsers allow for the inclusion of an external gadget in a web page in two different ways:

- either through the creation of a *script* node not subject to the Same Origin Policy, meaning that the included gadget is executed in the same environment as the integrator and has read/write access to all of its resources;

- or through the creation of an *iframe* node, subject to the Same Origin Policy, with the included gadget executed in a separate environment, commonly referred to as a *sandbox*, from which its does not have direct access to the integrator's resources[1].

Since the Same Origin Policy is, in fact, implemented in current browsers, it is possible to take advantage of it when designing secure web applications. This can be accomplished by the developer, as in [Barth 2009], or automatically, as in [Louw 2012] and [Luo 2012]. However, the complexities of the API for interframe communication often make it hard and cumbersome to manually sandbox the execution of external gadgets.

Even if we take advantage of the SOP to design secure web applications, we are still left with the problem of how to verify that a web application is in fact secure. Solving this problem is not trivial because even if we sandbox the execution a gadget (preventing it from **actively** compromising the integrity and confidentiality of the user's resources), the integrator can inadvertently leak confidential information to that gadget or corrupt high-integrity resources using data originating from that gadget. In other words, a sandboxing mechanism can allow the integrator to use the API for interframe communication as an *escape hatch* for sending/receiving **arbitrary** information to/from external gadgets. Hence, this type of mechanism is only fit to enforce security policies like *delimited release* [Sabelfeld 2003b], in which the integrator is allowed to declassify/endorse everything it sends/receives to/from external gadgets. In order to provide stronger security guarantees, one needs to resort to techniques more powerful than simply sandboxing the execution of third-party code. In particular, one needs to control the information flows that take place within the code of the integrator in order to decide which information can be securely sent to which gadget and/or which resources can be modified by which gadget-based information.

Another problem of SOP-based sandboxing mechanisms for web applications is that their precision is constrained by the precision of the SOP. In fact, it has been observed that *"the SOP is merely a highly restrictive Information Flow Control policy in which flows between origins are denied"* [Yang 2013]. By using the SOP as a means for securing web applications, one is essentially constraining the level of granularity of the security policies that can be enforced. Concretely, when using the SOP in the design of a security mechanism, one is forced to view each origin as a security principal in the system [Magazinius 2010]. Then, while it is possible to assign different security credentials to different sets of principals/origins, it is not possible to assign different security credentials to the same principal/origin depending on how it uses the information that it is given. For instance, suppose that we would like to express that a given gadget can have access to certain confidential information as long as it does not send it to the server from which it was issued. The only way to enforce this type of policy is through the use of an Information Flow Control (IFC) mechanism.

[1]In this case, communication is still possible via the PostMessage API [Barth 2009].

Just as the authors of [Yang 2013], we support the view that *"Information Flow Control is a good fit for whole-browser security"* as it can perfectly capture the SOP, but also express more fine-grained security policies whose enforcement may serve to eliminate security vulnerabilities in current web applications, while at the same time allowing for the flexibility of cross-origin communication.

1.1 Securing Information Flow in a Core of JavaScript

Noninterference [Goguen 1982] is a class of properties that reason about how the execution of a program propagates or generates dependencies between the resources it manipulates. The problem of enforcing secure information flow is essentially a problem of preventing the execution of programs that can potentially create illegal dependencies between the resources they operate on. For instance, confidentiality-wise, a program is secure if its execution does not entail the creation of dependencies between public outputs and secret inputs. In other words, public outputs cannot depend on secret inputs. Likewise, integrity-wise, a program is secure if if its execution does not entail the creation of dependencies between high-integrity outputs and low-integrity inputs. In other words, high-integrity outputs cannot depend on low-integrity inputs. Thus, noninterference provides the mathematical foundation for reasoning precisely about secure information flow and, in fact, it has been largely used [Hedin 2011, Sabelfeld 2003a] to formally express the absence of security leaks for a wide variety of programming languages ranging from functional (e.g. [Pottier 2003]) to object-oriented (e.g. [Banerjee 2002]) in both sequential (e.g. [Volpano 1996]) and concurrent settings (e.g. [Matos 2005]).

The stating of the dependencies that the execution of a program can legally generate generally betakes a certain degree of abstraction. It is not always possible or even desirable to talk about the actual resources that a program manipulates. Instead, it is often more convenient to reason about classes of resources that mandate the same degree of security. We can, therefore, see an information flow policy as a partially ordered set of security levels together with a mapping establishing the security levels of the resources on which the program operates. This mapping, which we call a *security labelling*, can be interpreted as an abstraction of the concrete resources of the program [Cousot 1977]. Having established a security policy, we say that an information flow between two given resources A and B is legal, if the security level of B is higher than or equal to the level of A. Whenever two levels L_A and L_B are in the order relation ($L_A \sqsubseteq L_B$), it means that the use of information at level L_B is at least as restrictive as the use of information at level L_A. More restrictive security levels correspond to higher confidentiality and lower integrity, since high-confidential resources are not allowed to affect low-confidential resources and low-integrity resources are not allowed to affect high-integrity resources. Intuitively, information is allowed to move up in the partially ordered set of security levels but not down. For convenience, we assume that the partially ordered set of security levels constitutes a lattice [Davey 2002], meaning that the *least upper bound* (*lub*) and the *greatest lower bound* (*glb*) between any two security levels are always defined.

In the context of information flow research, the enforcement of integrity policies [Biba 1977, Li 2003] can be viewed as the dual problem of the enforcement of confidentiality policies. Hence, in the remainder of the thesis we shall always refer to confidentiality policies, while the application of the proposed mechanisms to the enforcement of integrity policies would be straightforward.

Confidentiality-wise, given a concrete program state, a security labelling defines what part of that state is visible at each security level. Hence, if a security labelling is too coarse, it will declare invisible resources that should be visible. In this sense, coarse security policies inevitably cause secure programs not to abide by noninterference and therefore be rejected by sound enforcement mechanisms. Thus, it is vital that the *"abstractions made in the attacker model be adequate with respect to potential attacks"* [Sabelfeld 2003a]. In other words, security policies should be rich enough to capture the various types of attacks coming from the language, which means that they should adequately reflect its expressive power. The question to be answered is: *"What can an attacker see using the constructs of the language?"* The answer to this question is not always trivial, since not only are the contents of a program state visible to an attacker, but also the structure of these contents. For instance, in JavaScript, not only can a program see the values associated with the fields of an object, but it can also see the existence of any given field, and, therefore, its total number of fields.

In this thesis, we begin by defining noninterference for Core JavaScript - a fragment of JavaScript that retains its defining features. Particularly, the proposed definition of noninterference makes use of security policies that reflect the specificities of the language (such as the fact that programs can check the existence of object fields). We then study different types of mechanisms (both static and dynamic) to enforce variations of the proposed security property.

The dynamic nature of JavaScript renders it an exceedingly difficult language to analyse statically [Maffeis 2009]. Consequently, sound static analyses for JavaScript are in general largely over-conservative and reject many secure programs. Contrastingly, dynamic analyses are normally less conservative than static analyses, but impose a performance overhead that is often non-negligible [Hedin 2014]. In this thesis, we propose: **(1)** a purely dynamic monitor that enforces secure information flow in Core JavaScript as well as source-to-source transformation that inlines the monitor, **(2)** a type system that statically checks whether or not a Core JavaScript program abides by a given information flow policy, and finally **(3)** a hybrid type system that combines static and dynamic analyses in order to accept more programs than its fully static counterpart. This hybrid type system leverages the combination of static and runtime analysis to overcome some of the disadvantages of purely static and purely dynamic approaches.

1.2 Securing Information Flow in the Browser

Although JavaScript can be used as general-purpose programming language, most JavaScript programs are conceived to be executed in a browser in the context of a web page. These programs often interact with the web page in which they are included *via* the *Application Programming Interfaces* (APIs) provided by the browser, such as the Document Object Model API (DOM API), the XMLHttpRequest API, or the W3C Geolocation API. The semantics of these APIs often escapes the semantics of JavaScript in the sense that, since they are not implemented in JavaScript, their execution is not managed by the JavaScript engine, but rather by a dedicated and separate module of the browser [Grosskurth 2005]. Thus, a realistic analysis of client-side JavaScript code must include an analysis of the APIs that the targeted programs are supposed to use. However, the continuous emergence and heterogeneity of different APIs [Guha 2012] renders the problem of precise reasoning about JavaScript client-side code extremely challenging. This is particularly relevant in the context of information flow security. Hence, to tackle this problem, this thesis presents a general methodology for extending security monitors in order for them to take into account the possible invocation of arbitrary external APIs. We then apply this methodology to extend our information flow monitor for Core JavaScript as well as the corresponding source-to-source program transformation.

The DOM API [Recommendation 2000, Recommendation 2005] occupies a central role among the APIs that browsers make available for JavaScript programs. In fact, every modern browser includes a DOM implementation that manages the integration between JavaScript and the user interface of the browser. In other words, JavaScript programs use the DOM API to interact with the HTML page that the browser displays on the screen — to change or simply access the content of the page as well as the input coming from the user. In a certain sense, one can also view the DOM as the data structure corresponding to the “in memory” counterpart of the displayed HTML page. In fact, the displayed document is represented in the DOM as a tree structure. The nodes of the tree correspond to the various types of content in the document.

Unsurprisingly, malicious programs can use the DOM to encode illegal information flows [Russo 2009]. Hence, to make sure that a JavaScript program is secure, one must analyse how it interacts with the web page in which it is included via the DOM API. In this thesis, we present a group of monitor extensions for handling an important fragment of the DOM Core Level 1 API, that we call Core DOM. There, as in the DOM API, DOM nodes are treated as first-class values. Using this, we are able to construct an information flow control mechanism that is more fine-grained than the previous approaches in the literature [Russo 2009]. We also introduce methods and properties for modelling the behaviour of *live collections* — a special type of data structure in the DOM Core Level 1 API. We show that live collections effectively augment the observational power of an attacker and we show how to monitor their use in order to enforce secure information flow.

1.3 Contributions and Outline

In a nutshell, the original contributions of this thesis are the following:

- A new information flow monitor-inlining transformation for a core of JavaScript that retains its defining features;
- A hybrid type system for checking whether or not a Core JavaScript program abides by a given information flow policy that combines static and dynamic analysis to avoid rejecting programs that are in fact secure;
- A general methodology for extending information flow monitors to take into account the execution of arbitrary APIs, possibly outside of the perimeter of the modelled language;
- An information flow monitor that handles an important fragment of the DOM Core Level 1 API, including live collections, which had not been formally studied so far in the context of Information Flow Control (IFC) research.

The outline of the thesis is as follows:

- Chapter 2 presents the fragment of JavaScript that is studied in this thesis, which we call Core JavaScript. This core takes into account the defining features of the language, such as prototypical inheritance, extensible objects, constructs that check the existence of object fields, and atypical interactions between the binding of variables and the binding of object fields.
- Chapter 3 defines what it means for a Core JavaScript program to be noninterferent. The proposed definition of noninterference makes use of security policies that accurately capture the expressiveness of the language by taking into account its main specificities.
- Chapter 4 first presents a monitor that dynamically enforces secure information flow for Core JavaScript. Then, we define a source-to-source transformation that inlines the proposed monitor, and prove its correctness w.r.t. this monitor. Therefore, we ensure that, after compilation, only secure executions are allowed to go through, as potentially illegal executions are made divergent by the inlined runtime enforcement mechanism.

- Chapter 5 first presents and proves sound a purely static type system for securing information flow in Core JavaScript. Then, we present a hybrid version of this type system, which infers a set of assertions under which a program can be securely accepted and instruments it so as to dynamically check whether these assertions hold. By deferring rejection to runtime, this hybrid version is able to typecheck secure programs that purely static type systems cannot accept.

- Chapter 6 proposes a methodology for extending sound JavaScript information flow monitors. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, proving that a monitor is noninterferent after extending it with a new API only requires the proof that the API itself is noninterferent. We apply this methodology to extend our information flow monitor for Core JavaScript. Furthermore, this chapter presents an extension of the information flow monitor-inlining compiler defined in Chapter 4 that additionally takes into account the invocation of arbitrary APIs.

- Chapter 7 presents a group of monitor extensions for handling a fragment of the DOM Core Level 1 API, that we call Core DOM API. In the Core DOM API, as in the DOM API, tree nodes are treated as first-class values. We take advantage of this feature in order to design an information flow control mechanism that is more fine-grained than the previous approaches in the literature [Russo 2009]. Furthermore, we extend Core DOM with additional API methods that model the behaviour of *live collections*, a type of data structure present in the DOM Core Level 1 API that exhibits a very unusual semantics. We show that the use of live collections effectively augments the observational power of an attacker and we provide monitor extensions to tackle these newly introduced forms of information leaks.

1.4 Publications

While certain elements of this thesis remain unpublished to this day, the remaining parts have previously appeared in the following publications:

- Fragoso Santos, José and Rezk, Tamara. An Information Flow Monitor Inlining Compiler For Securing a Core of JavaScript. IFIP SEC, 2014
  This paper presents a version of the information flow monitor-inlining compiler here introduced in Chapter 4, which was, to the best of our knowledge, the first of this type of compilers designed for a JavaScript-like language. The information flow monitor used in the paper as well as its respective source-to-source transformation differ from those of the thesis in that they consider a smaller subset of JavaScript. Namely, they do not include neither the `in` nor the `delete` program constructs, which we do include here. Since the these constructs effectively augment the observational power of an attacker, their inclusion in the targeted fragment of the language required changing the way program resources are labeled.

- Almeida-Matos, Ana, Fragoso Santos, José and Rezk, Tamara. An Information Flow Monitor for a Core of DOM – Introducing references and live primitives. TGC, 2014
  In this paper, the authors propose and prove sound a novel, purely dynamic, flow-sensitive monitor for securing information flow in an imperative language extended with DOM-like tree operations. The monitor extensions presented in Chapter 6 partially coincide with the language primitives for operating on tree nodes studied in this paper. The main difference

is that here we study these operation in the context of Core JavaScript, while in the paper they were studied in the context of a simple WHILE language.

CHAPTER 2

Core JavaScript

Contents

In a nutshell, JavaScript is an object-oriented, untyped language which supports closures and prototype-based inheritance [5th edition of ECMA 262 June 2011 2011, 3rd edition of ECMA 262 1999, Crockford 2008, Flanagan 2011]. Indeed, objects are the central datatype of JavaScript. In contrast to class-based languages where the fields of an object are restricted by the class to which it belongs (which is statically specified), a JavaScript object is an unrestricted partial mapping from strings to values. The strings in the domain of an object are called its *properties*. There are no classes, but every (non-native) object has a prototype from which it can *inherit* properties. Prototypes are also objects. Hence, *prototypical inheritance* is a form of delegation, in the sense that an object dispatches to its prototype the requests that it does not know how to handle. For instance, in order to look-up the value of a property p of an object o, the JavaScript engine first checks whether p belongs to the set of properties of o. If so, the property look-up yields the value with which o associates property p, otherwise the engine checks whether the prototype of o defines a property named p, and so forth. The sequence of objects that can be accessed from a given object through the inspection of the respective prototypes is called a *prototype-chain*.

JavaScript features first-class functions. Functions can be used in three different ways: as usual functions, as *methods*, or as *constructors*. When assigning a function to a property of an object, the function becomes a *method* of the object. When calling a function as a method, the keyword `this` is bound to the receiver object. Every method accessible to an object through its prototype-chain can be called as a method of that object. For instance, if method `m` is accessible to object `o` through its prototype-chain, when calling `o.m(...)`, the keyword `this` is bound to `o` and not to the object that actually defines `m` in the prototype-chain of `o`. Hence, prototypes can be seen as a device for method sharing in JavaScript. Every function can additionally be called as a *constructor*. However, since we do not formally model the keyword `new`, we skip the explanation of this feature and refer the reader to [Flanagan 2011] for a detailed account of the language.

Another important feature of JavaScript is that programs are not only allowed to dynamically add new properties to the domain of an object, but they can also delete existing ones. A program can check whether a property is accessible from an object through its prototype-chain using the keyword `in`. Interestingly, the property look-up construct can also be used to check the existence of properties, since the looking-up of a property that is not defined in the prototype-chain of an object does not yield an error but instead a special value – `undefined`. Furthermore, the looking-up of a variable that has been declared but has not been yet assigned a value also

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

Where: e, e_0, e_1 and e_2 range over the set of expressions, x, y_1, ..., y_n range over the set of variable names, op ranges over the set of binary operators, and i and j range over the set of expression indexes.

Figure 2.1: Syntax of Core JavaScript

yields `undefined`. Besides `undefined`, JavaScript features another value meant to be used as a representation of no value – `null`. However, in contrast to `undefined`, `null` is an *assignment value*, meaning that it must be explicitly assigned to a variable/property so that its corresponding look-up yields `null`.

2.1 Syntax

We define a JavaScript-like language, called Core JavaScript, whose syntax is given in Figure 2.1. In Core JavaScript, some expressions are annotated with one or two unique indexes for the use of the semantics as well as the source-to-source transformations presented in the following chapters. We omit the index(es) of an expression whenever they are not needed. Furthermore, we use `o.p` as an abbreviation for `o["p"]`.

Core JavaScript is intended to model a realistic subset of the JavaScript specification [3rd edition of ECMA 262 1999]. However, in order to simplify the presentation, we do not model the `return` statement—functions are assumed to return the value to which their body evaluates. Furthermore, given that most implementations do allow explicit prototype mutation, we depart from [3rd edition of ECMA 262 1999] and include this feature through a special property `_prot_`. For instance, `o._prot_ = o_p` sets the prototype of `o` to `o_p`, and `o._prot_` evaluates to the prototype of `o`.

Figure 2.2 presents the running example that is used throughout the thesis. It consists of a fragment of the code for a simple contact management online application. The variable `CM` holds the *Contact Manager* object. The contact manager stores contacts in an object bound to its property `contact_list`, which is used as a table whose entries are the last names of the contacts (extended with unique integers to avoid collisions) and whose values are the actual contacts. A contact is simply an object containing a first name (stored in property `fst`), a last name (stored in property `lst`), an e-mail address (stored in property `email`), and a flag `favourite`. Observe that the mere existence of the property `favourite` in a contact object indicates by itself that that contact is among the user's favourite contacts. Therefore, the value assigned to this property is irrelevant and so we choose to always set it to `null`.

This example illustrates the typical use of prototypical inheritance in JavaScript. We create a "fixed" object bound to the property `proto_contact` of `CM` that stores all the methods contact objects are assumed to implement and every time a contact object is created, its prototype is set to `CM.proto_contact`. Hence, every contact object implements the methods: (1) `printContact` (that generates a string with a description of the contact), (2) `makeFavourite` (that marks the

```
CM = {}, CM.proto_contact = {}, CM.contact_list = {},

CM.proto_contact.printContact = function() { this.lst + "," + this.fst },

CM.proto_contact.makeFavourite = function() { this.favourite = null },

CM.proto_contact.unFavourite = function() {
   "favourite" in this ? delete this.favourite : true },

CM.proto_contact.isFavourite = function() { "favourite" in this },

CM.createContact = function(fst_name, lst_name, email) { var contact;
  contact = {}, contact._prot_ = proto_contact, contact.fst = fst_name,
  contact.lst = lst_name, contact.email = email, contact },

CM.storeContact = function(contact, i) {
   var list, key; list = this.contact_list, key = contact.lst+i,
   key in list ? CM.storeContact(contact, i+1) : list[key] = contact }

CM.getContact = function(lst_name, i) { this.contact_list[lst_name+i] }
```

Figure 2.2: A Simple Contact Manager

contact as favourite), (3) `isFavourite` (that checks whether the contact is marked as favourite), and (4) `unFavourite` (that deletes the property that marks the contact as favourite).

In the following, we give a brief description of the methods that compose the Contact Manager example. The method `printContact` simply returns a string consisting of the last and first names of the contact on which it was called separated by a comma (the binary operator $+$ should be interpreted as string concatenation). Since, the mere existence of the property `"favourite"` in a contact marks it as a *favourite* contact, the method `makeFavourite` only has to assign this property to an arbitrary value in order for the contact to become a *favourite* contact. Dually, in order for a contact to cease to be a favourite contact, one simply has to **delete** the property `"favourite"` from its list of properties. Finally, to check whether a contact is a favourite contact, one simply has to check whether `"favourite"` belongs to its list of properties. To do so, it suffices to use the program construct `in`. The method `createContact` creates a new contact and returns it. Therefore, the last expression in the body of this method is `contact`, since it evaluates to the newly created contact. Given a contact object and an integer `i`, the method `storeContact` first checks whether there already exists a contact with the same last name associated with `i` in the contact list, in which case the method calls itself recursively with the same contact and `i` incremented by one. Finally, the method `getContact` returns the contact associated with the name and integer that it receives as inputs. If no such contact exists, it will simply return `undefined`.

2.2 Formal Semantics

We model objects as partial functions mapping strings to values in a set $\mathcal{P}rim \cup \mathcal{R}ef \cup \mathcal{F}_\lambda$ containing all primitive values, references, and parsed function literals. The set $\mathcal{P}rim$ includes strings (taken from a set $\mathcal{S}tr$), numbers (taken from a set $\mathcal{N}um$), booleans (taken from a set $\mathcal{B}ool$), and two special values: `null` and `undefined`. References can be viewed as pointers to objects, in the sense that every expression that creates an object yields a new reference that points to it. As in [Banerjee 2002], we assume a *parametric object allocator*, meaning that references are chosen deterministically. While allowing us to some avoid technical complications in stating the main security property, it does not weaken the results of the thesis, since in practice allocators are in fact deterministic. The properties reserved for the internal use of the semantics are prefixed with an "@". We use $dom(o)$ for the set of properties of o excluding internal properties and $@dom(o)$ for the set of properties of o including internal properties. A memory $\mu : \mathcal{R}ef \mapsto \mathcal{S}tr \mapsto \mathcal{P}rim \cup \mathcal{R}ef \cup \mathcal{F}_\lambda$ is a mapping from references to objects [3rd edition of ECMA 262 1999]. Hence, given a memory μ and a reference r, $\mu(r)$ denotes the object bound to r in μ. Likewise, given an object o and a property p, $o(p)$ denotes the value bound to o's property p. Consequently, given a memory μ, a reference r, and a property p, $(\mu(r))(p)$ denotes the value bound to the property p of the object pointed to by r in μ. For simplicity, we use the notation $\mu(r \cdot p)$ as an abbreviation for $\mu(r)(p)$.

Before proceeding with the description of the formal semantics of Core JavaScript, we must introduce some auxiliary notation that is used throughout the thesis. We use: **(1)** $[p_0 \mapsto v_0, \cdots, p_n \mapsto v_n]$ for the partial function that maps p_0 to v_0, ..., and p_n to v_n respectively, **(2)** $f\,[p_0 \mapsto v_0, \cdots, p_n \mapsto v_n]$ for the function that coincides with f everywhere except in p_0, ..., p_n, which are otherwise mapped to v_0, ..., v_n respectively, and **(3)** $f|_P$ for the restriction of f to P (provided it is included in its domain). Furthermore, we use the notation $f[r \cdot p \mapsto p]$ as an abbreviation for the nested update $f\,[r \mapsto f(r)\,[p \mapsto v]]$.

In Core JavaScript, we model the binding of variables using *scope objects* [Maffeis 2008]. Hence, in the formal semantics, a function/method call triggers the creation of a scope object which maps its formal parameter as well as the variables declared in its body to their corresponding values. The creation of a scope object is formally emulated by the semantic relation $\mathcal{R}_{NewScope}$, which is given in Definition 2.1. If $\langle \mu, r_f, v_{arg}, r_{this}, i \rangle\ \mathcal{R}_{NewScope}\ \langle \mu', e, r' \rangle$, then: **(1)** μ' is the memory obtained from μ by the allocation of the new scope object in a new reference r', **(2)** r_f is the reference pointing to the function object whose code is to be executed, **(3)** e the body of the function, **(4)** v_{arg} the argument to be used, **(5)** r_{this} the reference pointing to the receiver object, and **(6)** i the index of the function/method call to be executed.

Definition 2.1 ($\mathcal{R}_{NewScope}$). *For any two memories μ and μ', three references r_f, r_{this}, and r', value v_{arg}, and expression e, $\langle \mu, r_f, v_{arg}, r_{this}, i \rangle\ \mathcal{R}_{NewScope}\ \langle \mu', e, r' \rangle$ holds if and only if:*

- $\lambda x.\{\mathsf{var}\ y_1, \cdots, y_n;\ e\} = \mu(r_f \cdot @code)$;
- $r = \mu(r_f \cdot @fscope)$;
- $r' = fresh(\mu, i)$;
- $\mu' = \mu\,[r' \mapsto [@fscope \mapsto r, x \mapsto v_{arg}, @this \mapsto r_{this}, y_1 \mapsto undefined, \cdots, y_n \mapsto undefined]]$

for some variables $x, y_1, \cdots, y_n$.

A scope object is said to be *active* if it is associated with the function/method that is currently executing. In order to handle scope composition, every scope object defines a property @*scope* that points to the scope object that was active when the corresponding function literal was evaluated. The sequence of scope objects that can be accessed from a given scope object through the respective @*scope* properties is called a *scope-chain*. The *global object*, which is assumed to be pointed to by a fixed reference $\#glob$, is the object that is at the end of every scope-chain and therefore it is the object that binds *global variables*. In particular, we assume that the global object also defines a property @*scope*, which in its case is set to *null*. In order to determine the value associated with a given variable, one has to inspect all objects in the scope-chain that starts in the *active* scope object. This behavior is modeled by the semantic relation $\mathcal{R}_{Scope}$ formally given in Definition 2.2. If $\langle \mu, r_0, x \rangle\ \mathcal{R}_{Scope}\ r_1$, then r_1 is the reference that points to the scope object that is closest to the one pointed to by r_0 in its corresponding scope-chain (whose objects are in the range of μ) and which defines a binding for variable x.

Definition 2.2 (Scope-Chain Inspection – $\mathcal{R}_{Scope}$). *The relation $\mathcal{R}_{Scope}$ is recursively defined as follows:*

$$\textsc{Null}\quad \langle \mu, null, x\rangle\ \mathcal{R}_{Scope}\ null \qquad \textsc{Base}\quad \frac{x \in dom(\mu(r))}{\langle \mu, r, x\rangle\ \mathcal{R}_{Scope}\ r} \qquad \textsc{Look-up}\quad \frac{x \notin dom(\mu(r)) \quad \langle \mu, \mu(r \cdot @scope), x\rangle\ \mathcal{R}_{Scope}\ r'}{\langle \mu, r, x\rangle\ \mathcal{R}_{Scope}\ r'}$$

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

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

where: **(1)** o_s^0, o_s^g, and o_s^h correspond to the scope objects associated with the invocation of the anonymous function, of function g, and of function h, respectively, **(2)** objects o_0, o_g, and o_h correspond to their respective function objects, and **(3)** $\hat{e}$ corresponds to the body of the anonymous function. After the execution of this program, the global object maps *out* to 0 and not to 1, because the scope object that is closest to o_s^h and which defines a binding for x is o_s^0 and not o_s^g (which does not belong to the scope-chain of o_s^h).

In Core JavaScript, every object (except scope objects and function objects) defines a property *_prot_* that stores a reference pointing to its prototype. The evaluation of an object literal yields a new reference, which is computed using the deterministic allocator $fresh$ and which is set to point to the newly created object. The property *_prot_* is originally set to *null*. When trying to look-up the value of a property p of an object o, the semantics first checks whether $p \in dom(o)$. If $p \in dom(o)$, the property look-up yields $o(p)$, otherwise the semantics checks whether the prototype of o (pointed to by $o(\_prot\_)$) defines a property named p, and so forth. The prototype-chain inspection procedure is emulated by the semantic relation $\mathcal{R}_{Proto}$ given in Definition 2.3. If $\langle \mu, r, m\rangle\ \mathcal{R}_{Proto}\ r'$, then r' is the closest reference to r in its corresponding prototype-chain (whose objects are in the range of μ) that defines a binding for m. Hence, the evaluation of $o_0 = \{\},\ o_0.p = 0,\ o_1 = \{\},\ o_1.\_prot\_ = o_0,\ o_1.p$ yields 0, because, although o_1 does not define property p, its prototype does. When looking-up the value of a property p in an object o, if p is not defined in the whole prototype-chain of o, instead of yielding an error, the semantics yields $undefined$. Therefore, the expression $o = \{\}, o.p$ evaluates to `undefined`.

Definition 2.3 (Prototype-Chain Inspection – $\mathcal{R}_{Proto}$). *The relation $\mathcal{R}_{Proto}$ is recursively defined as follows:*

$$\textsc{Null}\quad \langle \mu, null, m\rangle\ \mathcal{R}_{Proto}\ null \qquad \textsc{Base}\quad \frac{m \in dom(\mu(r))}{\langle \mu, r, m\rangle\ \mathcal{R}_{Proto}\ r} \qquad \textsc{Look-up}\quad \frac{m \notin dom(\mu(r)) \quad \langle \mu, \mu(r \cdot \_prot\_), m\rangle\ \mathcal{R}_{Proto}\ r'}{\langle \mu, r, m\rangle\ \mathcal{R}_{Proto}\ r'}$$

A function can be either invoked as a normal function or as a method. When calling a function as a method, the keyword this is bound to the receiver object, otherwise it is bound to the global object. Therefore, every scope object defines a property $@this$ (that was omitted in the first example) that holds the value of the keyword this in that scope. Hence, suppose that in a memory μ, the global object defines two variables o_0 and o_1 that hold references to the objects $[\_prot\_ \mapsto null, f \mapsto \#o_f]$ and $[\_prot\_ \mapsto \#o_0]$ respectively, where $\#o_f$ is the reference of a given function object. In the evaluation of expression $o_1.f(0)$, the semantics starts by creating a scope object in which property $@this$ is set to $\#o_1$ and then proceeds with the evaluation of the body of f.

In contrast to real client-side JavaScript where the global variable *window* holds a reference to the global object, in Core JavaScript a program cannot directly get hold of the reference pointing to the global object. However, any program can obtain a reference to the global object by evaluating the expression this in the body of a function called "as a function". For instance, after the evaluation of the program $x = 0$, $f = \mathsf{function}()\{\mathsf{this}\}$, $global = f()$, $global.x = 1$, the global variable x is bound to 1.

Figure 2.3 presents the big-step semantics for Core JavaScript. Every big-step semantic transition has the following form: $r \vdash \langle \mu, e \rangle \Downarrow \langle \mu', v \rangle$, where: **(1)** r is the reference of the active scope object, **(2)** μ and μ' are the initial and final memories, **(3)** e is the expression to evaluate, and **(4)** v is the value to which it evaluates. In the following, we give a brief description of the rules that better illustrate how the proposed semantics works:

- The Rule [Variable] starts by looking-up in the current scope-chain the reference of the scope-object that defines a binding for the variable x - r_x. Then, it returns the value with which that scope object associates x.

- The Rule [In Expression] starts by evaluating the two subexpressions of the current expression, thereby obtaining a reference to an object r_0 and a string name m_1. Then the semantics checks whether any of the objects in the prototype-chain of the object pointed to by r_0 defines a property named m_1. If that is the case, the expression evaluates to `tt`. Otherwise, it evaluates to `ff`.

- The Rule [Property Look-up] starts by evaluating the two subexpressions of the current expression, thereby obtaining the reference to the object whose property is being inspected (r_0) and the string corresponding to the property's name (m_1). Then, the semantics looks for the object that defines m_1 in the prototype-chain of the object pointed to by r_0. If that object exists, the semantics yields the value with which it associates property m_1. Otherwise, the semantics yields `undefined`.

- The Rule [Property Assignment] starts by evaluating the three subexpressions of the current expression, thereby obtaining the reference to the object whose property is being updated/created (r_0), the string corresponding to the property's name (m_1), and the value that is to be assigned to it (v_2). Then, the semantics sets the value of the property m_1 in the object pointed to by r_0 to v_2. This is done by setting r_0 to point to an object that coincides with $\mu_2(r_0)$ in every property except for m_1, which is set to point to v_2.

- The Rule [Method Call] starts by evaluating the three subexpressions of the current expression, thereby obtaining the reference to the object on which the method is called (r_0), the method's name (m_1), and the value to be used as an argument v_2. Then, the semantics finds the reference pointing to the object in the prototype-chain of the one pointed to by r_0 that actually implements the method named m_1 and obtains the function object corresponding to that method (r_f). Finally, the semantics allocates a new scope object and executes the body of the method.

- The Rule [Conditional Expression] starts by evaluating the guard of the conditional expression, thereby obtaining a value – $\hat{v}$. Then, the semantics whether $\hat{v}$ is a *falsy* value [Crockford 2008], that is whether $\hat{v} \in V_F = \{null, undefined, \mathtt{ff}, 0\}$. If $\hat{v}$ is not a *falsy* value, the then-branch of the conditional is executed. If it is, the else-branch is executed.

Value
$$r \vdash \langle \mu, v \rangle \Downarrow \langle \mu, v \rangle$$

This
$$r \vdash \langle \mu, \mathsf{this} \rangle \Downarrow \langle \mu, \mu(r \cdot @this) \rangle$$

Variable
$$\dfrac{\langle \mu, r, x \rangle \; \mathcal{R}_{Scope} \; r_x \qquad r_x \neq null}{r \vdash \langle \mu, x \rangle \Downarrow \langle \mu, \mu(r_x \cdot x) \rangle}$$

Binary Operation
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, v_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, v_1 \rangle \qquad v' = v_0 \; \mathsf{op} \; v_1}{r \vdash \langle \mu, e_0 \; \mathsf{op} \; e_1 \rangle \Downarrow \langle \mu_1, v \rangle}$$

Variable Assignment
$$\dfrac{r \vdash \langle \mu, e \rangle \Downarrow \langle \mu_0, v_0 \rangle \qquad \langle \mu_0, r, x \rangle \; \mathcal{R}_{Scope} \; r_x \qquad r_x \neq null \qquad \mu' = \mu_0[r_x \cdot x \mapsto v_0]}{r \vdash \langle \mu, x = e \rangle \Downarrow \langle \mu', v_0 \rangle}$$

Property Look-up
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, r_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, m_1 \rangle \qquad \langle \mu_1, r_0, m_1 \rangle \; \mathcal{R}_{Proto} \; r' \qquad r' \neq null \Rightarrow v = \mu_1(r' \cdot m_1) \qquad r' = null \Rightarrow v = undefined}{r \vdash \langle \mu, e_0[e_1] \rangle \Downarrow \langle \mu_1, v \rangle}$$

In Expression
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, m_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, r_1 \rangle \qquad \langle \mu_1, r_1, m_0 \rangle \; \mathcal{R}_{Proto} \; r' \qquad r' \neq null \Rightarrow v = \mathtt{ff} \qquad r' = null \Rightarrow v = \mathtt{tt}}{r \vdash \langle \mu, e_0 \; \mathsf{in} \; e_1 \rangle \Downarrow \langle \mu_1, v \rangle}$$

Property Assignment
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, r_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, m_1 \rangle \qquad r \vdash \langle \mu_1, e_2 \rangle \Downarrow \langle \mu_2, v_2 \rangle \qquad \mu' = \mu_2[r_0 \cdot m_1 \mapsto v_2]}{r \vdash \langle \mu, e_0[e_1] = e_2 \rangle \Downarrow \langle \mu', v_2 \rangle}$$

Property Deletion
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, r_0 \rangle \qquad \mu' = \mu_0 \left[r_0 \mapsto \mu_0(r_0)|_{dom(\mu_0(r_0)) \setminus \{p\}} \right]}{r \vdash \langle \mu, \mathsf{delete} \; e_0.p \rangle \Downarrow \langle \mu', \mathtt{tt} \rangle}$$

Function Call
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, r_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, v_1 \rangle \qquad \langle \mu_1, r_0, v_1, \#glob, i \rangle \; \mathcal{R}_{NewScope} \; \langle \hat{\mu}, \hat{e}, \hat{r} \rangle \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e} \rangle \Downarrow \langle \mu', v \rangle}{r \vdash \langle \mu, e_0(e_1)^i \rangle \Downarrow \langle \mu', v \rangle}$$

Method Call
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, r_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, m_1 \rangle \qquad r \vdash \langle \mu_1, e_2 \rangle \Downarrow \langle \mu_2, v_2 \rangle \qquad \langle \mu_2, r_0, m_1 \rangle \; \mathcal{R}_{Proto} \; r_m \qquad r_f = \mu_2(r_m \cdot m_1) \qquad \langle \mu_2, r_f, v_2, r_0, i \rangle \; \mathcal{R}_{NewScope} \; \langle \hat{\mu}, \hat{e}, \hat{r} \rangle \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e} \rangle \Downarrow \langle \mu', v \rangle}{r \vdash \langle \mu, e_0[e_1](e_2)^i \rangle \Downarrow \langle \mu', v \rangle}$$

Conditional Expression
$$\dfrac{r \vdash \langle \mu, \hat{e} \rangle \Downarrow \langle \hat{\mu}, \hat{v} \rangle \qquad \hat{v} \notin V_F \Rightarrow i = 0 \qquad \hat{v} \in V_F \Rightarrow i = 1 \qquad r \vdash \langle \hat{\mu}, e_i \rangle \Downarrow \langle \mu', v \rangle}{r \vdash \langle \mu, \hat{e} \; ? \; (e_0) : (e_1) \rangle \Downarrow \langle \mu', v \rangle}$$

Sequence
$$\dfrac{r \vdash \langle \mu, e_0 \rangle \Downarrow \langle \mu_0, v_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \rangle \Downarrow \langle \mu_1, v_1 \rangle}{r \vdash \langle \mu, e_0, e_1 \rangle \Downarrow \langle \mu_1, v_1 \rangle}$$

Object Literal
$$\dfrac{r' = fresh(\mu, i) \qquad \mu' = \mu \left[r' \mapsto [\_prot\_ \mapsto null] \right]}{r \vdash \langle \mu, \{\}^i \rangle \Downarrow \langle \mu', r' \rangle}$$

Function Literal
$$\dfrac{r' = fresh(\mu, i) \qquad \mu' = \mu \left[r' \mapsto [@fscope \mapsto r, \; @code \mapsto \lambda x. \{\mathsf{var} \; y_1, \cdots, y_n; \; e\}] \right]}{r \vdash \langle \mu, \mathsf{function}^i(x)\{\mathsf{var} \; y_1, \cdots, y_n; \; e\} \rangle \Downarrow \langle \mu', r' \rangle}$$

Figure 2.3: A Big-Step Semantics for Core JavaScript

2.3 Related Work

The popularity of JavaScript as a language for developing client-side web applications has been steadily increasing in recent years. This increase in popularity together with the theoretical challenges posed by the language have pushed forward a lot of research in both static and runtime analyses for JavaScript such as: type checking and type inference algorithms [Thiemann 2005, Anderson 2005, Jensen 2009], points-to analysis [Jang 2009], CPS-transformations [Luo 2012, Clements 2008] among others. Most of the analyses for JavaScript in the literature have been designed for different JavaScript-like languages, which capture different aspects of the real language. However, the great majority consists of a core lambda calculus extended with objects supporting prototype-based inheritance and imperative constructs. Some of these works also feature programming constructs for handling exceptions and implicit type coercions [Thiemann 2005].

Maffeis *et al* [Maffeis 2008] have been the first to propose a semantics for the full ECMA-262 Standard, 3rd Edition [3rd edition of ECMA 262 1999]. The proposed semantics is small-step and models the binding of variables using scope objects. More recently, Bodin *et al* [Bodin 2013] have presented a formalisation of the current version of the ECMA standard [5th edition of ECMA 262 June 2011 2011] in the Coq proof assistant as well as a JavaScript interpreter that has been proven correct with respect to the authors' specification. Furthermore, they have validated their interpreter using test262, the ECMA conformance test suite. In contrast to [Maffeis 2008], the formal semantics presented in [Bodin 2013] is big-step. This fact allows the authors to closely follow the informal specification, thereby maintaining what they call an *eyeball correspondence* between the standard and its formalisation in the Coq proof assistant. In order to overcome the typical drawbacks of big-step semantics (related to the handling of exceptions and divergence), the authors follow the *pretty big-step* style of Charguéraud [Charguéraud 2013]. Another important difference between these two semantics is that the authors of [Bodin 2013] model scope using *environment records* instead of scope objects. An environment record can be either a *declarative environment record* or an *object environment record*. While declarative environment records provide the local scoping associated with function calls, object environment records provide the dynamic scooping associated with the use of the construct `with`.

Also with the goal of reasoning precisely about real JavaScript programs, Guah *et al* [Guha 2010] have followed, however, a completely different approach from the works mentioned above. They have proposed λ_{JS} – a lambda calculus enriched with some of the most important JavaScript features, such as objects, prototype-based inheritance and constructs for handling exceptions, which the authors claim to capture the essence of JavaScript. Furthermore, they provide a de-sugaring transformation that compiles arbitrary JavaScript programs into λ_{JS} as well as an interpreter for λ_{JS} programs. These artefacts allowed them to validate their semantics and de-sugaring transformation by testing them against the test262 and Mozilla test suites.

2.4 Discussion

2.4.1 Modelling the Binding of Variables

JavaScript is not **statically scoped** in the sense that, in general, it is not possible to know statically in which scope we can find a property/variable. Consider, for instance, the following JavaScript program:

```
var x, y, obj0, obj1;
x = 0;
obj0 = {};
obj1 = {};
obj1.x = 1;
obj0._prot_ = obj1;
with(obj0) { y = x; }
```

After the execution of this program `y` is assigned to 1 and not to 0, because the `with` constructs adds `obj0` to the front of the current scope-chain, executes the assignment and then restores the scope-chain to its original state. Furthermore, since scope objects are allowed to have prototypes, the scope-chain inspection procedure traverses the prototype-chain of every scope object before going on to the next scope object. However, the current version of the specification [5th edition of ECMA 262 June 2011 2011] in *strict mode* is statically scoped, since it does not allow for the use of the most dynamic features of the language, such as the `with` construct.

Since scope objects are assumed not to have a prototype and since we do not include the JavaScript `with` construct, Core JavaScript programs are statically scoped. This means that we could have modelled the binding of variables using substitution, as in other works targeting subsets of the whole language, as [Guha 2010]. However, we have chosen to model scope using scope objects, as in [Maffeis 2008], for two main reasons. First, we envisage to extend the model to deal with a larger subset of the language, which may not be statically scoped. Second, modelling the binding of variables as the binding of properties allows us to simplify the definition of the security property for Core JavaScript.

Chapter 3

Defining Secure Information Flow in Core JavaScript

Contents

This chapter proposes a *noninterference* definition for Core JavaScript, which is in turn used to define what does it mean for a program to be secure. As a first step toward the definition of noninterference, we show how to label resources in Core JavaScript. Intuitively, a security labelling for a given memory establishes, for each security level, what parts of that memory are visible by an an attacker at that level. This is not easy to define since not only are the contents of the memory visible to an attacker, but also the structure of these contents. We use the term *security policy* for the pair consisting of a lattice of security levels and a security labelling. In the examples, we use the lattice $\mathcal{L} = \{H, L\}$ with $L \sqsubseteq H$ and $H \not\sqsubseteq L$, meaning that resources labeled with L (*low*) are less confidential than those labeled with H (*high*). Hence, H-labeled resources may depend on L-labeled resources, but not the contrary, as that would entail a *security leak*. We use $\sqcap$ and $\sqcup$ for the least upper bound (*lub*) and greatest lower bound (*glb*), respectively. And we use $\bot$ and $\top$ for the *bottom* level and the *top* level, respectively.

3.1 Challenges for IFC in Core JavaScript

Before proceeding to the formal definition of secure information flow in Core JavaScript, we review the main challenges imposed to information flow control by the particular features of the language. These challenges are particularly relevant to the definition of *security labelling* for a Core JavaScript memory.

3.1.1 Leaks via Prototype Mutations

The fact that a prototype of an object is allowed to change at runtime may be exploited to encode security leaks. For instance, returning to the example of the Contact Manager (given in Figure 2.2), suppose that the first and last names of a contact are of level L and that we create a new object, bound to `CM.proto_contact_new`, to be used as the prototype of contact objects, that prints contacts in a different way:

```
CM.proto_contact_new.printContact = function(){this.fst +"␣"+ this.lst}
```

The output of `printContact` is *low* for the original and new methods, since, in both cases, it only discloses information at level L. However, the expression:

```
h ? (c._proto_ = CM.proto_contact_new) : (null), l = c.printContact()
```

encodes an information flow from an H-labelled resource to an L-labelled resource because, depending on the value of the *high* variable `h`, it changes the prototype of `c` and therefore the behaviour of `printContact`, which is supposed to generate a *low* output. Concretely, depending on the value of `h`, the attacker sees the contact printed $last\_name, first\_name$ or $first\_name\ last\_name$. Hence, an IFC mechanism must be able to detect that the choice of which method to apply in the evaluation of `c.printContact()` effectively depends on H-labelled information.

3.1.2 Leaks via the Checking of the Existence of Properties

In Core JavaScript, a program can dynamically add and remove properties from objects. Furthermore, a program can check whether a property is defined in the prototype-chain of an object using the keyword `in`. Thus, the mere existence of a property in the domain of an object may disclose confidential information. As in [Hedin 2012], we associate every property in the domain of an object with an *existence level*. For instance, suppose that the user of the contact manager does not want to disclose which are his favorite contacts. In this case, the existence level of the property `favorite` must be set to H. However, the fact that a property is confidential does not imply that its existence is confidential. Suppose that the e-mail address associated with each contact is of level H. This does not mean that the existence level of the property `email` should be set to H. In fact, since all contact objects define a property `email` that is not supposed to be deleted, the existence of that property does not reveal any confidential information.

3.1.3 Leaks via the Global Object

During the execution of a function call, the keyword `this` is bound to the global object, whose properties are the global variables of the program. Hence, it is possible to encode illegal information flows regarding confidential global variables using the keyword `this` inside a function. For instance, the program `function() { l = this.cookie}()` produces the same effect as `l = cookie`. Dynamic IFC mechanisms are able to prevent this type of leak very simply, since it amounts to check whether the keyword `this` is bound to the global object. In contrast, static mechanisms for IFC face a much more difficult challenge, since it is very difficult to determine statically whether the `this` keyword may be bound to the global object in a given program point.

3.2 The Attacker Model

In order to formally characterize the "observational power" of an attacker, we take the standard approach of defining a notion of *low-projection* of a memory at a given level σ [Matos 2005], which corresponds to the part of the memory that an attacker at level σ can observe and which is given in Definition 3.1. To this end, we start by formally defining a security labelling as a tuple $\Sigma = \langle \Sigma_0, \Sigma_1, \Sigma_2 \rangle$ composed of three partial functions $\Sigma_0 : \mathcal{Ref} \mapsto \mathcal{L}$, $\Sigma_1 : \mathcal{Ref} \mapsto \mathcal{Str} \mapsto \mathcal{L}$, and $\Sigma_2 : \mathcal{Ref} \mapsto \mathcal{Str} \mapsto \mathcal{L}$ respectively called *object labelling*, *property-value labelling*, and *property-existence labelling* and such that:

- Σ_0 maps each reference in its domain to the security level associated with the object to which it points, called *object level*;
- Σ_1 maps each pair in its domain consisting of a reference and a property name to the security level associated with that property in the object pointed to by that reference, called *property-value level*;
- Σ_2 maps each pair in its domain consisting of a reference and a property name to the existence security level of that property in the object pointed to by that reference, called *property-existence level*.

Given a labelling Σ, we denote by $\Sigma.\mathsf{obj}$, $\Sigma.\mathsf{val}$, and $\Sigma.\mathsf{exist}$ the corresponding object labelling, property-value labelling, and property-existence labelling. Therefore, given an object o pointed to by a reference r, a labelling Σ, and a property name p: **(1)** $\Sigma.\mathsf{obj}(r)$ is the object level of o, **(2)** $\Sigma.\mathsf{val}(r)(p)$ is the property-value level of o's property p, and **(3)** $\Sigma.\mathsf{exist}(r)(p)$ is the property-existence level of o's property p. Informally, given a security labelling Σ, an attacker at level σ can see: **(1)** the existence of the

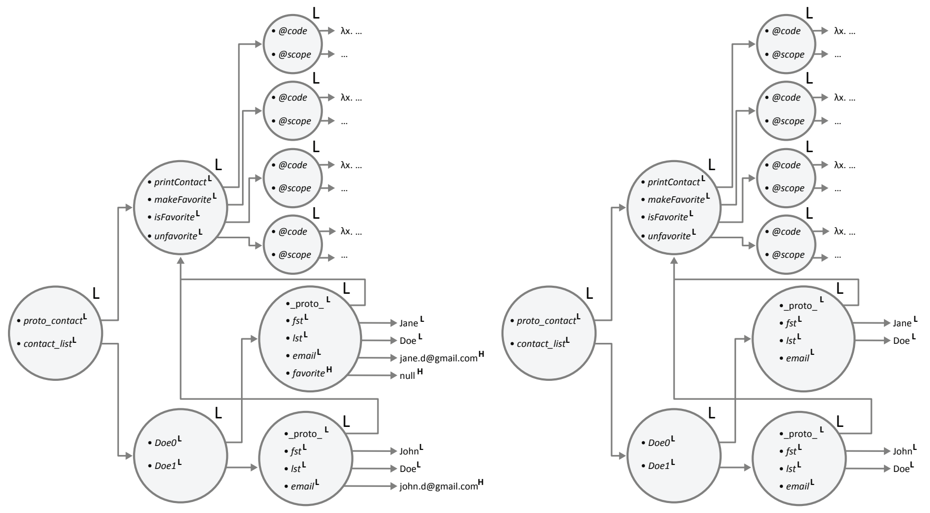


Figure 3.1: A memory and its low-projection

objects whose object levels are $\sqsubseteq \sigma$, **(2)** the existence of properties in visible objects whose property-existence levels are $\sqsubseteq \sigma$, and **(3)** the values associated with visible properties in visible objects whose property-value levels are $\sqsubseteq \sigma$.

Since not all program resources need to be labelled, a security labelling may be *partial*. However, there are some constraints it must verify. Namely, we say that memory μ is well-labeled by Σ if: **(1)** $dom(\Sigma.\mathsf{obj}) = dom(\Sigma.\mathsf{val}) = dom(\Sigma.\mathsf{exist}) \subseteq dom(\mu)$ and **(2)** for every reference $r \in dom(\Sigma.\mathsf{obj})$, $dom(\Sigma.\mathsf{val}(r)) = @dom(\Sigma.\mathsf{exist}(r)) \subseteq @dom(\mu(r))$.

Definition 3.1 (Low-Projection and Low-Equality for Core JavaScript Memories)**.** *The low-projection of a memory μ w.r.t. a security level σ and a labeling Σ is given by:*

$$\begin{aligned}\mu \restriction^{\Sigma,\sigma} = \; & \{(r, \Sigma.\mathsf{obj}(r)) \mid \Sigma.\mathsf{obj}(r) \sqsubseteq \sigma\} \\ \cup \; & \{(r, p, \Sigma.\mathsf{exist}(r)(p)) \mid \; \Sigma.\mathsf{obj}(r) \sqcup \Sigma.\mathsf{exist}(r)(p) \sqsubseteq \sigma \wedge p \in @dom(\mu(r))\} \\ \cup \; & \{(r, p, v, \Sigma.\mathsf{val}(r)(p)) \mid \; \Sigma.\mathsf{obj}(r) \sqcup \Sigma.\mathsf{exist}(r)(p) \sqcup \Sigma.\mathsf{val}(r)(p) \sqsubseteq \sigma \wedge p \in @dom(\mu(r))\}\end{aligned}$$

Two memories μ_0 and μ_1, respectively labeled by Σ_0 and Σ_1 are said to be low-equal at security level σ, written $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ if they coincide in their respective low-projections, $\mu_0 \restriction^{\Sigma_0,\sigma} = \mu_1 \restriction^{\Sigma_1,\sigma}$.

Returning to the Contact Manager example, suppose the user wants to enforce a security policy such that only the e-mails of the stored contacts and the identity of the *favourite* contacts should be of level H. Everything else should be set to L. Figure 3.1 presents the memory resulting from the execution of the program below:

```
x = CM.createContact("Jane", "Doe", "jane.d@gmail.com"),
y = CM.createContact("John", "Doe", "john.d@gmail.com"),
CM.storeContact(x, 0), CM.storeContact(y, 0), makeFavorite(x)
```

together with its low-projection at level L. Remark that, while the values of both e-mail addresses disappear, their existence remains visible. In contrast, the property `favourite` is removed from the contact object of Jane.

3.3 Noninterference for Core JavaScript

Informally, a program is *noninterferent* (NI) if its execution on two low-equal memories always produces two low-equal memories. Hence, an attacker cannot use a NI program as a means to disclose the confidential contents of a memory. It is convenient for subsequent chapters to start by defining the notion of a *noninterferent memory set* for a given program e. Intuitively, we say that a set of memories M is noninterferent w.r.t. a program e, an initial labelling Σ, and a final labelling Σ' at level σ, written $\mathbf{NI}^{\sigma}_{mem}(e, M, \Sigma, \Sigma')$, if the evaluation of e on any two memories in M that are low-equal memories according to Σ always produces two low-equal memories according to Σ'.

Definition 3.2 (Noninterferent Memory Set)**.** *A set M of memories is said to be a noninterferent memory set w.r.t. a program e, two labelings Σ and Σ', and a security level σ, written $\mathbf{NI}^{\sigma}_{mem}(e, M, \Sigma, \Sigma')$, if for any two memories $\mu_0, \mu_1 \in M$ such that: $\#glob \vdash \langle \mu_0, e \rangle \Downarrow \langle \mu_0', v_0 \rangle$, $\#glob \vdash \langle \mu_1, e \rangle \Downarrow \langle \mu_1', v_1 \rangle$, and $\mu_0, \Sigma \sim_\sigma \mu_1, \Sigma$, it holds that: $\mu_0', \Sigma' \sim_\sigma \mu_1, \Sigma'$.*

For simplicity, the definition of noninterferent memory set does not impose any restriction on the generated outputs. This does not constitute a problem, since any expression e that produces a *high* output can be trivially re-written as `h = e, null`.

Notice that it is possible to instantiate Definition 3.2 with an indistinguishability relation that allows indistinguishable memories to differ in their low parts (instead of instantiating it with the low-equality relation). For instance, one can use an indistinguishability criterion that only requires the average of all low variables in the two memories to coincide. In this way, we can characterize security properties that allow for declassification as shown in [Barthe 2011]. Definition 3.3 corresponds to the classical notion of noninterference. In the following we denote by $dom_{\Downarrow}(e)$ the set of memories on which the evaluation of e converges.

Definition 3.3 (Noninterference)**.** *A program e is noninterferent at level σ for two labelings Σ and Σ', written $\mathbf{NI}^{\sigma}(e, \Sigma, \Sigma')$ if and only if $\mathbf{NI}^{\sigma}_{mem}(e, dom_{\Downarrow}(e), \Sigma, \Sigma')$.*

3.4 Related Work

Since the seminal works of Bell and La Padula and Denning [Bell 1976, Denning 1976], the classical approach to secure information flow is to use a lattice of secure levels and a security labelling that maps resources to security levels. The ordering relation on the security levels establishes which are the legal information flows. Information is allowed to move up in the security lattice (from *low*-labelled resources to *high*-labelled resources), but not down. This property was first formally stated via a notion *strong dependency* by Cohen in [Cohen 1977], and later referred to as *noninterference* by Goguen and Messeguer in [Goguen 1982].

In general, one can view *noninterference* as a class of properties that state how the execution of a program is allowed to propagate dependencies between the resources on which it operates. In order to instantiate noninterference to a concrete programming language, one must start by defining how to label program states. While simple imperative languages only require a very simple labelling strategy [Volpano 1996], more complex languages may require sophisticated labelling strategies whose details heavily depend on the features of the targeted language.

Hedin *et al* [Hedin 2012] have been the first to propose an information flow monitor for a realistic core of JavaScript. They introduce the notion of *existence levels* to deal with the constructs for the checking of the existence of properties. They further introduce the notion of *structure security level* (SSL), which corresponds to an upper bound on the existence levels of the properties of an object. Hence, if an object o has a *low* SSL, one can only change its structure (either by adding properties to o or removing properties from o) in low contexts.

3.5 Discussion

3.5.1 Towards an Attacker Model for the Ecma standard

The attacker model we present here fits the expressiveness of Core JavaScript. The Ecma standard [5th edition of ECMA 262 June 2011 2011], however, allows for other types of attacks. Namely, in JavaScript, an attacker can explore time-based covert channels [Agat 2000] to encode illegal information flows, which is not the case in Core JavaScript. Consider, for instance, the program below:

```
l1 = (new Date()).getTime();
if (h) {
   // do meaningless time-consuming operations
}
l2 = (new Date()).getTime() - l1
```

where the expression `new Date()` evaluates to an object that represents the current date, which, in turn, implements a method `getTime` that outputs the time in milliseconds since 1970/01/01. After the execution of this program, the value of `l2` depends on the initial value of the *high* variable `h`. Therefore, information flow control mechanisms targeting the full Ecma standard must be able to detect these types of flows.

3.5.2 Further Remarks about the Structure Security Level

It is important to emphasise that the *structure security level* [Hedin 2012] is not a key element for the characterisation of the attacker model inherent to JavaScript, but rather a device of the authors' enforcement mechanism. The need for the SSL arises from the fact that the existence levels are not established *a priori*. Hence, the SSL plays the role of the existence level of the properties that do not exist yet. Accordingly, the level associated with the look-up of a property that does not exist is the SSL. Consider the example: $\texttt{o} = \{\}$, $\texttt{h}$? $(\texttt{o.p} = 0)$: $(\texttt{null})$, $\texttt{l} = \texttt{p in o}$. The monitor of Hedin *et al* will either raise the level of `l` to H (if the SSL of o is *high*) or block the assignment (if the SSL of o is *low*).

Chapter 4

Dynamic Information Flow Control in Core JavaScript

Contents

Due to the dynamic nature of JavaScript, research on mechanisms to check the compliance of JavaScript programs with noninterference has mostly focused on dynamic approaches, such as information flow monitors [Austin 2012, Hedin 2012] and secure multi-execution [Devriese 2010]. In practice, there are two main approaches for implementing a JavaScript information flow monitor: either one modifies a JavaScript engine so that it additionally implements the security monitor (as in [Hedin 2012]), or one inlines the monitor in the original program (as in [Magazinius 2012, Chudnov 2010]). The second approach, which we follow, has the advantage of being *browser-independent*. This chapter presents a compiler that inlines an information flow monitor for Core JavaScript.

The proposed compiler is proven sound w.r.t. a standard definition of input-output termination insensitive noninterference for monitors. Informally, we prove that the execution of a compiled program only goes through if it is noninterferent; otherwise, the constraints inlined in the program by the compiler cause it to diverge. The chapter is divided into two main sections. Section 4.1 presents an information flow monitored semantics for Core JavaScript that is proven *sound*, i.e. proven to enforce termination-insensitive noninterference. The proposed monitored semantics differs from a previous monitor for enforcing secure information flow in a realistic core of JavaScript [Hedin 2012] in that it was specifically designed to guide the implementation of an inlining compiler rather than a browser instrumentation. Section 4.2 presents an inlining compiler that rewrites Core JavaScript programs in order to simulate their execution in the monitor. The compiler is proven *correct*, meaning that the execution of a program goes through in the monitor *if and only if* the execution of its instrumentation by the inlining compiler goes through in the original semantics. In order to this, the security labelling is instrumented in the program's memory, thus giving raise to a *similarity relation* between *labelled memories* and *instrumented memories*. As illustrated in Figure 4.1, given a labelled memory and its instrumented counterpart, the monitored execution of the original program in the labelled memory and the standard execution of its compilation in the instrumented memory always yield two similar memories. We have implemented a prototype of the proposed compiler, which supports a subset of JavaScript semantics larger than the one modelled in Core JavaScript.

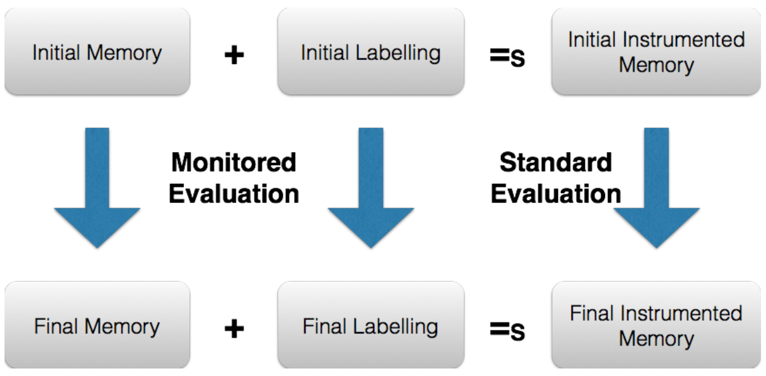


Figure 4.1: Monitored Execution of Program vs. Unmonitored Execution of Compilation

4.1 Monitoring Secure Information Flow in Core JavaScript

In this section, we present a monitored semantics for dynamically enforcing secure information flow in Core JavaScript. The security monitor we present is flow-sensitive, purely dynamic and follows the *no-sensitive-upgrade* discipline of Zdancewic [Zdancewic 2002, Austin 2009].

The monitored execution of an expression e in a memory μ paired up with a security labelling Σ can be interpreted as an extension of the unmonitored execution of e in μ that additionally performs the *abstract execution* of e in Σ. In this sense, we can view Σ as an abstract memory. While the standard execution of e in μ produces a value, its abstract execution in Σ generates a security level σ, which is called the *reading effect* of e [Sabelfeld 2003a]. The reading effect of e corresponds to the least upper bound on the levels of the resources on which the value to which e evaluates depends. The rules of the monitored semantic relation, $\Downarrow_{IF}$, are defined in Figure 4.3. The semantic rules have the form $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$, where: **(1)** σ_{pc} is *the security level of the program counter*, that is, the security level of the current execution context **(2)** Σ and Σ' are the initial and final security labellings, and **(3)** σ is the *reading effect* of e. All the remaining elements keep their original meaning. For simplicity, the monitor was designed in such a way that the reading effect of an expression is always higher than or equal to the level of the context in which it was evaluated. For clarity, in the specification of each semantic rule, we use:

- light grey for the parts of the rule that coincide with the unmonitored semantics,
- orange for the labelling updates,
- red for the constraints.

The security level associated with checking the existence of a given property in a given object is the corresponding *property-existence level.* It is natural for a dynamic enforcement mechanism to set the existence level of a given property to the level of the context in which it was created. This, however, raises the problem of deciding which is the existence level of the properties that do not exist yet. For instance, suppose a program checks whether an object o defines a given property p. If p is in the domain of o, the security level of the result should be the existence level of p. But what if p is not in the domain of o? To cope with this issue, each object is associated with a *default existence level* that acts as the existence level of the properties that do not exist yet and which is called *structure security level* [Hedin 2012]. Hence, in the previous example, when p is not in the domain of o, the level of the result should be the structure security level of o.

To keep track of the structure security levels of the objects in memory, dynamic security labelings are extended with a fourth element, called *structure security labelling*, that maps each object reference to the structure security level of the corresponding object. Furthermore, in this chapter, we assume that object literals are annotated with their corresponding structure security levels. Given a security labelling Σ, we denote by $\Sigma.\mathsf{struct}$ the corresponding structure security labelling.

In order to ease the specification of the monitor, we introduce a group of functions to update security labellings, which are presented in Figure 4.2 and which we briefly describe below:

$$\textsc{Labelling Update}\quad \frac{\begin{array}{c} p \in dom(\Sigma.\mathsf{exist}(r)) \Rightarrow \Sigma_{exist} = \Sigma.\mathsf{exist} \\ p \notin dom(\Sigma.\mathsf{exist}(r)) \Rightarrow \Sigma_{exist} = \Sigma.\mathsf{exist}[r \cdot p \mapsto \sigma] \\ \Sigma_{val} = \Sigma.\mathsf{val}[r \cdot p \mapsto \sigma'] \end{array}}{updt(\Sigma, (r, p), (\sigma, \sigma')) = \langle \Sigma.\mathsf{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma.\mathsf{struct} \rangle}$$

$$\textsc{Labelling Contraction}\quad \frac{\begin{array}{c} P = @dom(\Sigma.\mathsf{exist}(r)) \backslash \{p\} \qquad \Sigma_{val} = \Sigma.\mathsf{val}\,[r \mapsto \Sigma.\mathsf{val}(r)|_P] \\ \Sigma_{exist} = \Sigma.\mathsf{exist}\,[r \mapsto \Sigma.\mathsf{exist}(r)|_P] \end{array}}{contract(\Sigma, r, p) = \langle \Sigma.\mathsf{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma.\mathsf{struct} \rangle}$$

$$\textsc{Labelling Extension}\quad \frac{\begin{array}{cc} \Sigma_{obj} = \Sigma.\mathsf{obj}\,[r \mapsto \sigma_o] & \Sigma_{val} = \Sigma.\mathsf{val}\,[r \mapsto [\ ]] \\ \Sigma_{exist} = \Sigma.\mathsf{exist}\,[r \mapsto [\ ]] & \Sigma_{struct} = \Sigma.\mathsf{struct}\,[r \mapsto \sigma_s] \end{array}}{extend(\Sigma, r, \sigma_o, \sigma_s) = \langle \Sigma_{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma_{struct} \rangle}$$

Figure 4.2: Meta-Functions to Update Security Labellings

- $updt(\Sigma, (r, p), (\sigma, \sigma'))$ outputs the security labelling obtained from Σ by setting the value level of the property p in the object pointed to by r to σ'. Furthermore, if this object does not already define a property p, the existence level of p is set to σ.
- $contract(\Sigma, r, p)$ outputs the security labelling obtained from Σ by removing the existence level and the value level of the property p in the object pointed to by r.
- $extend(\Sigma, r, \sigma_o, \sigma_s)$ outputs the security labelling obtained from Σ when allocating a new object with level σ_o and structure security level σ_s.

In the following we give a brief description of the rules of the monitored semantics. We ignore by now some important aspects of the monitor, such as the constraints that it enforces, which are carefully discussed in the following subsections. As a general remark, if a rule does not change the memory, it also does not change the security labelling.

- [Value] The reading effect of a value is simply the level of the program counter.
- [This] The reading effect of the expression this is the *lub* between the level of the program counter and the *value level* of the internal property $@this$ in the current scope object.
- [Variable] The reading effect of a variable x is the *lub* between the level of the program counter and the *value level* of the property x in the scope object that defines a binding for x in the current scope-chain.
- [Binary Operation] The reading effect of a binary operation e_0 op e_1 is simply the *lub* between the reading effects of e_0 and e_1. It is important to emphasise that both the reading effect of e_0 and the reading effect of e_1 are already higher than or equal to the level of the program counter. Hence, the reading effect of e_0 op e_1 is also higher than or equal to the level of the program counter.
- [Variable Assignment] The reading effect of a variable assignment $x = e_0$ is simply the reading effect of e_0, which is already higher than or equal to the level of the program counter. This rule also sets the *value level* of the property x in the scope object that defines a binding for x in the current scope-chain to the reading effect of e_0. The *existence level* of x in that scope-object remains unchanged, because x is already supposed to be there. The constraint of this rule, as all the other constraints, is explained in Subsection 4.1.1.
- [Property Look-up] The reading effect of a property look-up $e_0[e_1]$ is the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure (explained

in Subsection 4.1.3), and **(3)** the *value level* of the property m_1 (obtained from the evaluation of e_1) in the object that defines a binding for it in the prototype-chain of the object pointed to by r_0 (obtained from the evaluation of e_0), **provided that such object exists.**

- [In Expression] The reading effect of an in-expression e_0 in e_1 is the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure (explained in Subsection 4.1.3), and the *existence level* of m_0 (obtained from the evaluation of e_0) in the object that defines a binding for it in the prototype-chain of the object pointed to by r_1 (obtained from the evaluation of e_1), **provided that such object exists.**

- [Property Assignment] The reading effect of a property assignment $e_0[e_1] = e_2$ is simply the reading effect of e_2. This rule also sets the *value level* of property m_1 (obtained from the evaluation of e_1) in the object pointed to by r_0 (obtained from the evaluation of e_0) to the *lub* between the reading effects of the **three** subexpressions. If the property assignment is a property creation (meaning that m_1 is not already defined by the object pointed to by r_0), the existence level of m_1 in the object pointed to by r_0 is set to the *lub* between the reading effects of e_0 and e_1.

- [Property Deletion] The reading effect of a property deletion delete $e.p$ is simply the level of the program counter, as a property deletion does not reveal any information about its subexpressions. This rule also removes both the *value level* and the *existence level* of the property p in the object pointed to by r (obtained from the evaluation of e).

- [Function Call] The reading effect of a function call $e_0(e_1)$ is the reading effect of the body of the function that is evaluated. The allocation of the new scope object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated scope object. This extension is discussed in detail in Subsection 4.1.4. The level of the program counter during the evaluation of the body of the function is set to the *lub* between the reading effect of e_0 and the level of the context in which the corresponding function literal was evaluated.

- [Method Call] The reading effect of a method call $e_0[e_1](e_2)$ is the reading effect of the body of the method that is evaluated. Like in the case of the function call, the allocation of the new scope object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated scope object. The level of the program counter during the evaluation of the body of the method is set to the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure, **(3)** the level of the context in which the function literal corresponding to the method was evaluated, and **(4)** the *value level* of the property m_1 (obtained from the evaluation of e_1) in the object that defines a binding for m_1 in the prototype-chain of the object pointed to by r_0 (obtained from the evaluation of e_0).

- [Object Literal] The reading effect of an object literal is simply the level of the program counter. The allocation of the new object must be paired-up with an extension of the current labelling in order to record the *object level* and the *structure security level* of the object. Furthermore, it must also record the *value level* and the *existence level* of the property *_prot_* of the newly allocated object, which are both set to the current level of the program counter.

- [Function Literal] The reading effect of a function literal is simply the level of the program counter. The allocation of the new function object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated function object: *@fscope* and *@code*. The *value level* and the *existence level* of both of these properties are set to the current level of the program counter.

- [Conditional Expression] The reading effect of a conditional expression is the reading effect of the branch that is evaluated. During the evaluation of this branch, the level of the program counter is upgraded to the reading effect of the guard of the conditional. Hence, the reading effect of whole conditional expression is always higher than or equal to the reading effect of its guard.

- [Sequence] The reading effect of a sequence expression e_0, e_1 is the reading effect of its second subexpression.

Value

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

This

$$\frac{r_{this} = \mu(r \cdot @this) \qquad \sigma_{this} = \Sigma.\mathsf{val}(r \cdot @this) \sqcup \sigma_{pc}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{this}, \Sigma \rangle \Downarrow_{IF} \langle \mu, r_{this}, \Sigma, \sigma_{this} \rangle}$$

Variable

$$\frac{\begin{array}{c} \langle \mu, r, x \rangle \ \mathcal{R}_{Scope}\ r_x \qquad r_x \neq null \\ \sigma = \Sigma.\mathsf{val}(r_x \cdot x) \sqcup \sigma_{pc} \end{array}}{r, \sigma_{pc} \vdash \langle \mu, x, \Sigma \rangle \Downarrow_{IF} \langle \mu, \mu(r_x \cdot x), \Sigma, \sigma \rangle}$$

Binary Operation

$$\frac{\begin{array}{c} \forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ v' = v_0 \ \mathsf{op}\ v_1 \qquad \sigma' = \sigma_0 \sqcup \sigma_1 \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0 \ \mathsf{op}\ e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_2, v', \Sigma_2, \sigma' \rangle}$$

Variable Assignment

$$\frac{\begin{array}{c} r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle \qquad \langle \mu_0, r, x \rangle \ \mathcal{R}_{Scope}\ r_x \qquad r_x \neq null \\ \mu' = \mu_0[r_x \cdot x \mapsto v_0] \qquad \Sigma' = updt(\Sigma_0, (r_x, x), (\Sigma_0.\mathsf{exist}(r_x \cdot x), \sigma_0)) \qquad \sigma_{pc} \sqsubseteq \Sigma_0.\mathsf{val}(r_x \cdot x) \end{array}}{r, \sigma_{pc} \vdash \langle \mu, x = e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu', v_0, \Sigma', \sigma_0 \rangle}$$

Property Look-up

$$\frac{\begin{array}{c} \forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle \mu_2, v_0, v_1, \Sigma_2 \rangle \ \mathcal{R}_{Proto}\ \langle r', \sigma' \rangle \qquad \sigma'' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma' \\ r' = null \Rightarrow v = undefined \ \wedge\ \sigma = \sigma'' \\ r' \neq null \Rightarrow v = \mu_1(r' \cdot m_1) \ \wedge\ \sigma = \sigma'' \sqcup \Sigma.\mathsf{val}(r' \cdot v_1) \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1], \Sigma \rangle \Downarrow_{IF} \langle \mu_2, v, \Sigma_2, \sigma \rangle}$$

In Expression

$$\frac{\begin{array}{c} \forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle \mu_2, v_0, v_1, \Sigma_2 \rangle \ \mathcal{R}_{Proto}\ \langle r', \sigma' \rangle \qquad \sigma'' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma' \\ r' = null \Rightarrow v = \mathtt{ff} \ \wedge\ \sigma = \sigma'' \\ r' \neq null \Rightarrow v = \mathtt{tt} \ \wedge\ \sigma = \sigma'' \sqcup \Sigma.\mathsf{exist}(r' \cdot v_1) \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0 \ \mathsf{in}\ e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_2, v, \Sigma_2, \sigma \rangle}$$

Property Assignment

$$\frac{\begin{array}{c} \forall_{i=0,1,2}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \qquad v_0 \in \mathcal{R}ef \qquad v_1 \in \mathcal{S}tr \\ \mu' = \mu_3[v_0 \cdot v_1 \mapsto v_2] \qquad \Sigma' = updt(\Sigma_3, (v_0, v_1), (\sigma_0 \sqcup \sigma_1, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2)) \\ v_1 \in dom(\mu_3(v_0)) \Rightarrow \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_3.\mathsf{val}(v_0 \cdot v_1) \qquad v_1 \notin dom(\mu_3(v_0)) \Rightarrow \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_3.\mathsf{struct}(v_0) \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1] = e_2, \Sigma \rangle \Downarrow_{IF} \langle \mu', v_2, \Sigma', \sigma_2 \rangle}$$

Property Deletion

$$\frac{\begin{array}{c} r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \rangle \qquad p \in dom(\mu_0(r_0)) \\ \mu' = \mu_0\left[r_0 \mapsto \mu_0(r_0)|_{dom(\mu_0(r_0)) \setminus p}\right] \\ \Sigma' = contract(\Sigma_0, r_0, p) \qquad \sigma_0 \sqsubseteq \Sigma_0.\mathsf{exist}(r_0 \cdot p) \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, \mathsf{delete}\ e_0.p, \Sigma \rangle \Downarrow_{IF} \langle \mu', \mathtt{tt}, \Sigma', \sigma_{pc} \rangle}$$

Function Call

$$\frac{\begin{array}{c} \forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle \mu_2, v_0, v_1, \#glob, i, \Sigma_2, \sigma_0, \sigma_1 \rangle \ \mathcal{R}_{NewScope}\ \langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}, \hat{\sigma}_{pc} \rangle \\ \hat{r}, \hat{\sigma}_{pc} \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0(e_1)^i, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle}$$

Method Call

$$\frac{\begin{array}{c} \forall_{i=0,1,2}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle \mu_3, v_0, v_1, \Sigma_3 \rangle \ \mathcal{R}_{Proto}\ \langle r_m, \sigma_m \rangle \qquad r_f = \mu_3(r_m \cdot v_1) \\ \sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma_3.\mathsf{val}(r_m \cdot v_1) \sqcup \sigma_m \\ \langle \mu_3, r_f, v_2, v_0, i, \Sigma_3, \sigma_0, \sigma_1 \rangle \ \mathcal{R}_{NewScope}\ \langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}, \hat{\sigma}_{pc} \rangle \\ \hat{r}, \hat{\sigma}_{pc} \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle \end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1](e_2)^i, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle}$$

Object Literal

$$\frac{\begin{array}{c} r_o = fresh(\mu, i) \\ \mu' = \mu\left[r_o \mapsto [\_prot\_ \mapsto null]\right] \\ \Sigma' = extend(\Sigma, r_o, \sigma_{pc}, \sigma_s) \\ \Sigma'' = updt(\Sigma', (r_o, \_prot\_), (\sigma_{pc}, \sigma_{pc})) \end{array}}{r, \sigma_{pc} \vdash \langle \mu, \{\}^{i, \sigma_s}, \Sigma \rangle \Downarrow_{IF} \langle \mu', r_o, \Sigma'', \sigma_{pc} \rangle}$$

Function Literal

$$\frac{\begin{array}{c} r_f = fresh(\mu, i) \qquad \mu' = \mu\left[r' \mapsto [@fscope \mapsto r, @code \mapsto \lambda x.\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]\right] \\ \Sigma' = extend(\Sigma, r_o, \sigma_{pc}, [@fscope \mapsto (\sigma_{pc}, \sigma_{pc}), @code \mapsto (\sigma_{pc}, \sigma_{pc})], \sigma_s) \end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}, \Sigma \rangle \Downarrow_{IF} \langle \mu', r_f, \Sigma', \sigma_{pc} \rangle}$$

Conditional

$$\frac{\begin{array}{c} r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \hat{\mu}, \hat{v}, \hat{\Sigma}, \hat{\sigma} \rangle \\ \hat{v} \notin V_F \Rightarrow i = 0 \qquad \hat{v} \in V_F \Rightarrow i = 1 \\ r, \sigma_{pc} \sqcup \hat{\sigma} \vdash \langle \hat{\mu}, e_i, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle \end{array}}{r, \sigma_{pc} \vdash \langle \mu, e\ ?\ (e_0) : (e_1), \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle}$$

Sequence

$$\frac{\begin{array}{c} r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle \\ r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle \end{array}}{r, \sigma_{pc} \vdash \langle \mu, e_0, e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle}$$

Figure 4.3: Monitored Core JavaScript Semantics

Program:	$h = 0$	$h = 1$	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
$l_0 = \texttt{tt}$;	$\Sigma.\mathsf{val}(l_0) := L$	$\Sigma.\mathsf{val}(l_0) := L$	$\Sigma.\mathsf{val}(l) := L$
$l_1 = \texttt{tt}$;	$\Sigma.\mathsf{val}(l_1) := L$	$\Sigma.\mathsf{val}(l_1) := L$	$\Sigma.\mathsf{val}(l) := L$
h ?	branch not taken	branch taken	branch taken
($l_0 = \texttt{ff}$);	—	$\Sigma.\mathsf{val}(l_0) := H$	*stuck*
l_0 ?	branch taken	branch not taken	—
($l_1 = \texttt{ff}$);	$\Sigma.\mathsf{val}(l_1) := L$	—	—
Final Low Memory:	$l_1 = \texttt{ff}$	$l_1 = \texttt{tt}$	—

Table 4.1: Naive Approach vs No-sensitive-upgrade

4.1.1 Controlling Implicit Flows and the No-Sensitive-Upgrade Discipline

The *no-sensitive-upgrade* discipline of Zdancewic [Zdancewic 2002, Austin 2009] establishes that visible resources cannot be upgraded in invisible contexts, since such upgrades cause the visible domain of a program to change depending on secret values. Hence, flow-sensitive monitors that implement the no-sensitive-upgrade discipline abort executions that encode illegal implicit flows. Intuitively, one could consider a *naive* strategy that would simply raise the security level of visible resources updated in *high* contexts to the level of the context itself. However, this strategy does not work since it partially leaks the contents of the resources on which the control flow depends. Consider, for instance, the example given in Table 4.1 and adapted from [Austin 2010]. This table shows four monitored executions of a program (represented on the left) in two distinct memories that initially map a *high* variable h to 0 and 1, respectively. Specifically, one can see how the dynamic labelling Σ evolves during the execution of the program applying both the naive strategy and the no-sensitive-upgrade strategy. While both monitors coincide on the executions starting from the memory that initially maps h to 0, they differ on the executions starting from the memory that initially maps h to 1. The monitor following the *naive* approach raises the level of l_0 to H (thus allowing the execution to go through), whereas the monitor following the *no-sensitive-upgrade* strategy blocks the execution when the program tries to update the value of l_0 in a high context. Observe that the execution of this program by the monitor following the *naive* strategy generates two memories that are **not** low-equal even though the initial memories are low-equal.

In Core JavaScript there are six types of implicit illegal flows, illustrated in Table 4.4, that cause the proposed monitor to abort the execution. To see why the information flows encoded in the programs given in Table 4.4 should be prevented, consider their execution by a monitor following the *naive* approach in two memories that initially map a *high* variable h to 0 and 1, respectively. The execution of all six programs in a memory that originally maps h to 0 terminates with a memory that maps the low variable l to `ff` (without raising its security level to H). Alternatively, their execution in a memory that originally maps h to 1 terminates with a memory that maps the low variable l to `tt` (without raising its security level to H). Since the two initial memories are low-equal, one can see that the execution of these programs by a monitor following the naive strategy reveals information about the secret contents of the initial memory (specifically, the content of the *high* variable h). Below, we list and briefly comment each type of illegal implicit flow:

- **Visible Variable Assignment in Invisible Context (Type I):** the monitor blocks assignments to variables holding visible values in *high* contexts. Therefore, in the example, the monitor blocks the assignment of `ff` to l_{aux} inside the first conditional.
- **Visible Property Assignment in Invisible Context (Type II):** the monitor blocks assignments to properties holding visible values within invisible contexts. Therefore, in the example, the monitor blocks the assignment of `ff` to $o.p$ inside the first conditional.
- **Visible Property Deletion in Invisible Context (Type III):** the monitor blocks deletions of

Type I	**Type II**	**Type III**
$l_{aux} = \texttt{tt}$,	$o = \{\}$,	$o = \{\}$,
$l = \texttt{tt}$,	$o.p = \texttt{tt}$,	$o.p = \texttt{tt}$,
$h\ ?\ (l_{aux} = \texttt{ff})$,	$l = \texttt{tt}$,	$l = \texttt{tt}$,
$l_{aux}\ ?\ (l = \texttt{ff})$	$h\ ?\ (o.p = \texttt{ff})$,	$h\ ?\ (\mathsf{delete}\ o.p)$,
	$o.p\ ?\ (l = \texttt{ff})$	"p" in $o\ ?\ (l = \texttt{ff})$

Type IV	**Type V**	**Type VI**
$o_h = \{\}$,	$l = \texttt{tt}$,	$o_h = \{\}$,
$o_l = \{\}$,	$o = \{\}$,	$o_l = \{\}$,
$l = \texttt{tt}$,	$o.q = \texttt{ff}$,	$o_h.p = \texttt{tt}$,
$o_l.p = \texttt{ff}$,	$prop_h =$ "p",	$o_l.p = \texttt{tt}$,
$h\ ?\ (o_h = o_l)$,	$h\ ?\ (prop_h =$ "q"$)$,	$l = \texttt{tt}$,
$o_h.p = \texttt{tt}$,	$o[prop_h] = \texttt{tt}$,	$h\ ?\ o_h = o_l$,
$!o_l.p\ ?\ (l = \texttt{ff})$,	$!o.q\ ?\ (l = \texttt{ff})$	$\mathsf{delete}\ o_h.p$,
		"p" in $o_l\ ?\ (l = \texttt{ff})$

Table 4.2: Naive Approach vs No-sensitive-upgrade

visible properties in invisible contexts. Therefore, in the example, the monitor blocks the deletion of o's property p inside the first conditional.

- **Visible Property Assignment via Invisible Reference (Type IV):** the monitor blocks assignments to visible properties when the reference pointing to the object that binds the property was computed using secret information. For instance, in the example, while the *low* variable o_l can only hold *low* references, the *high* variable o_h can hold both *low* references and *high* references. Therefore, the assignment $o_h = o_l$ is allowed to go through. However, when o_h is set to point to the same reference as o_l, the assignment $o_h.p = \texttt{tt}$ is blocked, since it tries to update the value of a *low* property via a *high* reference.
- **Visible Property Assignment via Invisible Property Name (Type V):** the monitor blocks assignments to visible properties when the corresponding property name was computed using secret information. For instance, in the example, the variable $prop_h$ can hold both *low* and *high* property names. Therefore, the assignment $prop_h =$ "q" is allowed to go through, even though it is performed inside a *high* conditional. However, after this assignment, the assignment $o[prop_h] = \texttt{tt}$ is blocked since it tries to update the value of a *low* property via a *high* property name.
- **Visible Property Deletion via Invisible Reference (Type VI):** the monitor blocks visible property deletions when the reference pointing to the object that binds the property was computed using secret information. For instance, in the example, the *high* variable o_h can hold both *low* references and *high* references. Therefore, the assignment $o_h = o_l$ is allowed to go through. However, when o_h is set to point to the same reference as o_l, the execution of $\mathsf{delete}\ o_h.p$ is blocked since it constitutes a *low* property deletion via a *high* reference.

4.1.2 The Structure Security Level

Since in Core JavaScript objects are initially created without any properties, the structure security level of an object defines an upper bound for the existence levels of the properties that can be added to that object. In this sense, *the structure security level of an object can be understood as the security level associated with its domain.* It is important to emphasise that the structure security level is not a key

Program:	$h = 0$	$h = 1$	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
$l = \texttt{tt}$;	$\Sigma.\mathsf{val}(l) := L$	$\Sigma.\mathsf{val}(l_0) := L$	$\Sigma.\mathsf{val}(l) := L$
$o = \{\}^L$;	$\Sigma.\mathsf{val}(o) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$	$\Sigma.\mathsf{val}(o) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$	$\Sigma.\mathsf{val}(o) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$
h ?	branch not taken	branch taken	branch taken
$(o.p = \texttt{tt})$;	—	$\Sigma.\mathsf{val}(r_o, \text{“}p\text{”}) := H/$ $\Sigma.\mathsf{exist}(r_o, \text{“}p\text{”}) := H/$ $\Sigma.\mathsf{struct}(r_o) := L$	*stuck*
$!(\text{“}p\text{”} \ \mathsf{in} \ o)$?	branch taken	branch not taken	—
$(l = \texttt{ff})$;	$\Sigma.\mathsf{val}(l) := L$	—	—
Final Low Memory:	$l = \texttt{ff}$	$l = \texttt{tt}$	—

Table 4.3: Preventing Security Leaks via the Domain of an Object

element for the characterisation of the attacker model inherent to JavaScript, but rather a device of the enforcement mechanism. The need for the structure security level arises from the fact that existence levels are not established *a priori*.

Since the structure security level is used to control the **implicit information flows** that can be encoded by modifying the domain of an object, it cannot be upgraded. In fact, such upgrades would violate the no-sensitive-upgrade discipline, which forbids upgrades based on implicit flows. Hence, if an object o has a *low* structure security level, one can only change its structure (either by adding properties to o or removing properties from o) in *low* contexts.This fact is illustrated in Table 4.3, which shows four monitored executions of a program in two distinct memories that initially map a *high* variable h to 0 and 1 respectively. While both monitors coincide on the executions starting from the memory that initially maps h to 0, they differ on the executions starting from the memory that initially maps h to 1. The monitor following the *naive* approach raises the structure security level of the object bound to o to H (thus allowing the execution to go through), whereas the monitor following the *no-sensitive-upgrade* strategy blocks the execution when the program tries to create a property in an object with a *low* structure security level within a *high* context. We assume in this example that the created object is stored in reference r_o. Observe that the execution of this program by the monitor following the *naive* strategy generates two memories that are **not** low-equal even though the initial memories are low-equal.

4.1.3 Preventing Security Leaks via Prototype Mutations

When a program looks up the value of a property p in an object o, if $p \notin dom(o)$, the security level associated with the property look-up expression must be equal to or higher than the structure security level of o, because this property look-up leaks information about o's domain. Concretely, one gets to know that p does not belong to the domain of o. Furthermore, it must also be higher than or equal to the level of o's property *_prot_*, since the value of this property determines what is the object that the prototype-chain look up procedure will inspect next. In fact, the security monitor has to take into account the structure security level as well as the level of property *_prot_* of every object traversed during the prototype-chain inspection procedure until it finds the object that defines a binding for the property being looked-up. For example, given a memory:

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

and a labeling Σ, such that either $\Sigma.\mathsf{struct}(\#o_0) = H$ or $\Sigma.\mathsf{val}(\#o_0 \cdot \_prot\_) = H$, the reading effect of the expression $o_1.p$ must be H, because it leaks information about the domain of o_1 and about the prototype of o_1. We redefine (in Definition 4.1) the prototype-chain look-up procedure in order to additionally compute the security level associated with the prototype-chain inspection procedure.

Definition 4.1 ($\mathcal{R}_{Proto}$). *The relation $\mathcal{R}_{Proto}$ is recursively defined as follows:*

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

$$\textsc{Look-up}\quad \frac{\begin{array}{c} m \notin dom(\mu(r)) \qquad r' = \mu(r \cdot \_prot\_) \\ \langle \mu, r', m, \Sigma\rangle\ \mathcal{R}_{Proto}\ \langle r'', \sigma\rangle \\ \sigma' = \Sigma.\mathsf{val}(r \cdot \_prot\_) \sqcup \Sigma.\mathsf{struct}(r) \sqcup \sigma \end{array}}{\langle \mu, r, m, \Sigma\rangle\ \mathcal{R}_{Proto}\ \langle r', \sigma'\rangle}$$

4.1.4 Tracking the Level of the Program Counter

An information flow monitor must keep track of *the level of the program counter* in order prevent illegal implicit flows. In the particular case of Core JavaScript, the level of the program counter must always be higher than or equal to the security levels of the resources that were used to decide:

- which branch to take in a conditional expression whose code is still executing,
- which function/method to execute in a function/method call expression whose whose code is still executing.

In order to account for the first point, when evaluating a branch of a conditional expression, the level of the program counter is upgraded to the reading effect of its guard. Handling the second point is not as easy. When calling a function/method, the level of the program counter must be upgraded to the *lub* between:

- the reading effects of the expressions that were used to decide which function/method to call,
- the level of the context in which the function literal corresponding to the function/method that is to be executed was evaluated.

In order to illustrate the **first point**, consider the following expression:

```
f1 = function(x) { l = 0 },
f2 = function(x) { l = 1},
h ? (f = f1) : (f = f2),
f()
```

Assuming that the security level of `h` is originally set to *high*, we conclude that the security level of `f` must also be set to *high*. Otherwise, the monitor aborts the execution of the conditional (regardless of the taken branch). Therefore, the level of the program counter must be also set to *high* during the execution of the function bound to `f`. This causes the monitor to abort the execution, since both functions perform *low* assignments. To illustrate the **second point**, consider the expression:

```
f = h ? (function (x) { l = 0 }) : (function (x) { l = 0 }),
f()
```

After the evaluation of the conditional expression, `f` is bound to a function object corresponding to a function literal that was evaluated in a *high* context. Therefore, during the execution of the function bound to `f`, the level of the program counter must be set to *high*, which renders the *low*-assignment performed in its body illegal.

Given a function f whose function object is pointed to by r_f, the monitored semantics book-keeps the level of the context in which the function literal corresponding to f was evaluated in: $\Sigma.\mathsf{val}(r_f \cdot @fscope)$. Definition 4.2 modifies the semantic function $\mathcal{R}_{NewScope}$, introduced in Definition 2.1, for it to additionally capture the extension of the security labelling to the newly created scope object. Hence, if $\langle \mu, r_f, v_{arg}, r_{this}, i, \Sigma, \sigma_{pc}, \sigma_{arg}\rangle\ \mathcal{R}_{NewScope}\ \langle \mu', e, r', \Sigma', \sigma'_{pc}\rangle$, then: **(1)** Σ' is the labelling obtained from Σ by covering the newly allocated scope object, **(2)** σ_{pc} is the level of the context in which the function was called, **(3)** σ_{arg} is the level of the argument, and **(4)** σ'_{pc} is the level of the context in which the function body is to be executed. The remaining elements keep their previous interpretation.

Definition 4.2 ($\mathcal{R}_{NewScope}$). *For any two memories μ and μ', three references r_f, r_{this}, and r', value v_{arg}, and expression e, $\langle \mu, r_f, v_{arg}, r_{this}, i, \Sigma, \sigma_{pc}, \sigma_{arg}\rangle\ \mathcal{R}_{NewScope}\ \langle \mu', e, r', \Sigma', \sigma'_{pc}\rangle$ holds if and only if:*

- $\lambda x. \{\mathsf{var}\ y_1, \cdots, y_n;\ e\} = \mu(r_f \cdot \textit{@code})$;
- $r = \mu(r_f \cdot \textit{@fscope})$;
- $r' = \textit{fresh}(\mu, i)$;
- $\mu' = \mu\,[r' \mapsto [\textit{@fscope} \mapsto r, x \mapsto v_{arg}, \textit{@this} \mapsto r_{this}, y_1 \mapsto \textit{undefined}, \cdots, y_n \mapsto \textit{undefined}]]$;
- $\sigma'_{pc} = \sigma_{pc} \sqcup \Sigma.\mathsf{val}(r_f \cdot \textit{@fscope})$;
- $\Sigma'.\mathsf{obj} = \Sigma.\mathsf{obj}\left[r' \mapsto \sigma'_{pc}\right]$;
- $\Sigma'.\mathsf{exist} = \Sigma.\mathsf{exist}\left[r' \mapsto \left[\textit{@fscope} \mapsto \sigma'_{pc}, x \mapsto \sigma'_{pc}, \textit{@this} \mapsto \sigma'_{pc}, y_1 \mapsto \sigma'_{pc}, \cdots, y_n \mapsto \sigma'_{pc}\right]\right]$;
- $\Sigma'.\mathsf{val} = \Sigma.\mathsf{val}\left[r' \mapsto \left[\textit{@fscope} \mapsto \sigma'_{pc}, x \mapsto \sigma'_{pc} \sqcup \sigma_{arg}, \textit{@this} \mapsto \sigma'_{pc}, y_1 \mapsto \sigma'_{pc}, \cdots, y_n \mapsto \sigma'_{pc}\right]\right]$;
- $\Sigma'.\mathsf{struct} = \Sigma.\mathsf{struct}\left[r' \mapsto \sigma'_{pc}\right]$

for some variables $x, y_1, \cdots, y_n$.

4.1.5 Security Guarantees - Soundness

We say that a security monitor is *noninterferent if and only if* monitored executions always preserve the low-equality relation. Informally, an information flow monitor is noninterferent *if and only if*, for any expression e, whenever an attacker cannot distinguish two labeled memories before executing e, then the attacker is also unable to distinguish the final memories. Hence, an attacker cannot use the monitored execution of a program as a means to disclose information about the confidential contents of a memory. Theorem 4.1 states that the monitored successfully-terminating execution of a program on two low-equal memories always yields two low-equal memories.

Theorem 4.1 (Noninterferent Monitor). *For any expression e, memories μ and μ', respectively labeled by Σ and Σ', reference r, and security levels σ_{pc} and σ, such that:*

- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$,
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$,
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$;

Then: $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ *and if either* $\sigma_f \sqsubseteq \sigma$ *or* $\sigma'_f \sqsubseteq \sigma$, *then* $v_f = v'_f$.

The second claim of the theorem states that, whenever one of the executions produces a visible value, the other also produces a visible value and the two values coincide.

4.1.5.1 Establishing Noninterference

Preliminaries Before stating the results that need to be established in order to prove that the proposed monitor is noninterferent, we start by defining low-equality for labeled values. Informally, two values v_0 and v_1 respectively labelled by σ_0 and σ_1 are said to be low-equal at level σ, written $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$ if either they are both observable and coincide or they are both unobservable. Formally: $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$ if and only if: $v_0 = v_1 \ \wedge\ \sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma \ \vee\ \sigma_0 \sqcap \sigma_1 \not\sqsubseteq \sigma$. This equation can be equivalently re-written as $(\sigma_0 \sqsubseteq \sigma \ \vee\ \sigma_1 \sqsubseteq \sigma) \Rightarrow (\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma \ \wedge\ v_0 = v_1)$.

Proving Confinement Classically, one of the first steps towards proving a noninterference result is to establish a *confinement result*. In the present case, Theorem 4.2 establishes that the monitored execution of a Core JavaScript expression in a *high* context does **not** update or create *low* memory. Therefore, when executing a Core JavaScript program using the monitor in a *high* context, the low-projections of the initial and final memories coincide.

Theorem 4.2 (Confinement). *Given an expression e, a memory μ, a labelling Σ, a level σ_{pc} and a reference r such that: $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$ for some memory μ', value v, labelling Σ' and security level σ; then for every security level $\sigma' \in \mathcal{L}$ such that $\sigma_{pc} \not\sqsubseteq \sigma'$: $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$.*

In order to prove confinement, one first needs to state for each type of operation that possibly modifies the memory, the conditions under which that type of operation is *confined*. We identify four types of operations that change the memory:

- **Property Assignment.** A property assignment changes the memory either by creating a new property in an existing object or by updating the value of an existing property of an existing object. Lemma 4.1 states that a **property update** is confined if the *value level* of the updated property is not observable, whereas a **property creation** is confined if the *existence level* of the created property is not observable.
- **Property Deletion.** A property deletion changes the memory by deleting an existing property in an existing object. Lemma 4.2 states that a property deletion is confined if the *existence level* of the deleted property is not observable.
- **Object Creation.** Lemma 4.3 states that an object creation is confined if both the the object level and the structure security level of the created object are not observable.
- **Scope Allocation.** Lemma 4.4 states that the allocation of a scope object is not observable as long as the level of the context in which the body of the function that is to be executed is not observable.

Lemma 4.1 (Confined Property Assignment)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference r, a property p, a value v, and three security levels $\sigma, \sigma', \sigma'' \in \mathcal{L}$, such that:* **(1)** $\mu' = \mu[r \cdot p \mapsto v]$, **(2)** $\Sigma' = \mathit{updt}(\Sigma, (r,p), (\sigma', \sigma''))$, *and* **(3)** $p \notin \mathit{dom}(\mu(r)) \Rightarrow \sigma' \not\sqsubseteq \sigma$, *and* **(4)** $p \in \mathit{dom}(\mu(r)) \Rightarrow \sigma'' \not\sqsubseteq \sigma$*; then, it follows that* $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.

Lemma 4.2 (Confined Property Deletion)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference r, a property p, and a security level $\sigma \in \mathcal{L}$, such that:* **(1)** $\mu' = \mu\left[r \mapsto \mu(r)|_{dom(\mu(r))\setminus p}\right]$, **(2)** $\Sigma' = \mathit{contract}(\Sigma, r, p)$, *and* **(3)** $\Sigma.\mathsf{exist}(r \cdot p) \not\sqsubseteq \sigma$*; then, it follows that* $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.

Lemma 4.3 (Confined Object Creation)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference $r \notin \mathit{dom}(\mu)$, and three security levels $\sigma, \sigma_o, \sigma_s \in \mathcal{L}$, such that:* **(1)** $\mu' = \mu\,[r \mapsto [\_prot\_ \mapsto \mathit{null}]]$, **(2)** $\Sigma' = \mathit{updt}(\Sigma'', (r, \_prot\_), (\sigma_o, \sigma_o))$ *where* $\Sigma'' = \mathit{extend}(\Sigma, r, \sigma_o, \sigma_s)$, *and* **(3)** $\sigma_o \sqcap \sigma_s \not\sqsubseteq \sigma$*; then, it follows that* $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.

Lemma 4.4 (Confined Scope Allocation)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', three references $r_f, r_{this}, r_{scope} \in \mathcal{R}ef$, a value v_{arg}, an integer i, and four security levels $\sigma, \sigma_{arg}, \sigma_{pc}, \sigma'_{pc} \in \mathcal{L}$, such that:* **(1)** $\langle \mu, r_f, v_{arg}, r_{this}, i, \Sigma, \sigma_{pc}, \sigma_{arg} \rangle \; \mathcal{R}_{NewScope} \; \langle \mu', e, r_{scope}, \Sigma', \sigma'_{pc} \rangle$ *and* **(2)** $\sigma'_{pc} \not\sqsubseteq \sigma$*; then, it follows that* $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.

Proving Noninterference In order to prove noninterference, it is useful to establish some intermediate results about the outcome of applying read/write operations in low-equal memories in observable contexts. We start by establishing two indistinguishability results concerning the scope-chain and the prototype-chain look-up procedures. Concretely, Lemma 4.5 states that the results of applying the scope-chain look-up procedure in two low-equal memories in visible scopes are the same. Lemma 4.6 states that the results of applying the prototype-chain look-up procedure in two low-equal memories are low-equal. That is, either both results are observable and coincide or they are both unobservable.

Lemma 4.5 (Scope-Chain Indistinguishability)**.** *Given two memories μ_0 and μ_1 respectively labelled by Σ_0 and Σ_1, a reference r, a security level σ, and a string $m \in \mathcal{S}tr$ such that:* **(1)** $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$, **(2)** $\langle \mu_0, r, m \rangle \; \mathcal{R}_{Scope} \; r_0$, **(3)** $\langle \mu_1, r, m \rangle \; \mathcal{R}_{Scope} \; r_1$, *and* **(4)** $\Sigma_0.\mathsf{obj}(r) \sqcup \Sigma_1.\mathsf{obj}(r) \sqsubseteq \sigma$*; it follows that:* $r_0 = r_1$.

Lemma 4.6 (Prototype-Chain Indistinguishability)**.** *Given two memories μ_0 and μ_1 respectively labelled by Σ_0 and Σ_1, a reference r, a security level σ, and a string $m \in \mathcal{S}tr$ such that:* **(1)** $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$, **(2)** $\langle \mu_0, r, m, \Sigma_0 \rangle \; \mathcal{R}_{Proto} \; \langle r_0, \sigma_0 \rangle$, *and* **(3)** $\langle \mu_1, r, m, \Sigma_1 \rangle \; \mathcal{R}_{Proto} \; \langle r_1, \sigma_1 \rangle$*; it holds that:* $r_0, \sigma_0 \sim_\sigma r_1, \sigma_1$.

Finally, one needs to state for each type of operation that possibly modifies the memory, the conditions under which, when performed in low-equal memories in low-equal contexts, they produce low-equal memories. As we did for confinement, we consider each type of write-operation individually.

- **Property Assignment.** Lemma 4.7 states that the assignment of two low-equal values to the same property of two objects pointed to by the same reference in two low-equal memories yields two low-equal memories.
- **Property Deletion.** Lemma 4.8 states that the deletion of the same property in two objects pointed to by the same reference in two low-equal memories yields two low-equal memories.
- **Object Creation.** Lemma 4.9 states that the allocation of a new empty object in the same new reference in two low-equal memories yields two low-equal memories.
- **Scope Allocation.** Lemma 4.10 states that the allocation of a new scope object in the same new reference in two-equal memories yields two low-equal memories.

Lemma 4.7 (Noninterferent Property Assignment)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, a reference r, a property p, two values v_0 and v_1, and four security levels $\sigma, \sigma', \sigma_0, \sigma_1 \in \mathcal{L}$, such that:*

- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0[r \cdot p \mapsto v_0]$ *and* $\mu'_1 = \mu_1[r \cdot p \mapsto v_1]$,
- $\Sigma'_0 = \mathit{updt}(\Sigma_0, (r, p), (\sigma', \sigma_0))$ *and* $\Sigma'_1 = \mathit{updt}(\Sigma_1, (r, p), (\sigma', \sigma_1))$,
- $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$;

then, it follows that $\mu'_0, \Sigma'_0 \sim_\sigma \mu'_1, \Sigma'_1$.

Lemma 4.8 (Noninterferent Property Deletion)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, a reference r, a property p, and a security level σ, such that:*

- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0 \left[r \mapsto \mu_0(r)|_{dom(\mu_0(r))\backslash p}\right]$ *and* $\mu'_1 = \mu_1 \left[r \mapsto \mu_1(r)|_{dom(\mu_1(r))\backslash p}\right]$,
- $\Sigma'_0 = \mathit{contract}(\Sigma_0, r, p)$ *and* $\Sigma'_1 = \mathit{contract}(\Sigma_1, r, p)$;

then, it follows that $\mu'_0, \Sigma'_0 \sim_\sigma \mu'_1, \Sigma'_1$.

Lemma 4.9 (Noninterferent Object Creation)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, and three security levels $\sigma, \sigma_o, \sigma_s \in \mathcal{L}$, such that:*

- $r \notin dom(\mu_0) \cup dom(\mu_1)$,
- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0\left[r \mapsto [\_prot\_ \mapsto null]\right]$ *and* $\mu'_1 = \mu_1\left[r \mapsto [\_prot\_ \mapsto null]\right]$,
- $\Sigma'_0 = \mathit{updt}(\Sigma''_0, (r, \_prot\_), (\sigma_o, \sigma_o))$ *and* $\Sigma'_1 = \mathit{updt}(\Sigma''_1, (r, \_prot\_), (\sigma_o, \sigma_o))$,

where $\Sigma''_0 = \mathit{extend}(\Sigma_0, r, \sigma_o, \sigma_s)$ *and* $\Sigma''_1 = \mathit{extend}(\Sigma_1, r, \sigma_o, \sigma_s)$*; then, it follows that* $\mu'_0, \Sigma'_0 \sim_\sigma \mu'_1, \Sigma'_1$.

Lemma 4.10 (Noninterferent Scope Allocation)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, three references $r_f, r_{this}, r_{scope} \in \mathcal{R}ef$, two values v^0_{arg} and v^1_{arg}, an integer i, and four security levels $\sigma, \sigma^0_{arg}, \sigma^1_{arg}, \sigma_{pc}, \hat{\sigma}^0_{pc}, \hat{\sigma}^1_{pc} \in \mathcal{L}$, such that:*

- $v^0_{arg}, \sigma^0_{arg} \sim_\sigma v^1_{arg}, \sigma^1_{arg}$
- $\langle \mu_0, r_f, v^0_{arg}, r_{this}, i, \Sigma_0, \sigma_{pc}, \sigma^0_{arg} \rangle \ \mathcal{R}_{NewScope} \ \langle \mu'_0, e_0, r^0_{scope}, \Sigma'_0, \hat{\sigma}^0_{pc} \rangle$,
- $\langle \mu_1, r_f, v^1_{arg}, r_{this}, i, \Sigma_1, \sigma_{pc}, \sigma^1_{arg} \rangle \ \mathcal{R}_{NewScope} \ \langle \mu'_1, e_1, r^1_{scope}, \Sigma'_1, \hat{\sigma}^1_{pc} \rangle$,

then, it follows that $\mu'_0, \Sigma'_0 \sim_\sigma \mu'_1, \Sigma'_1$ *and either* $\hat{\sigma}^0_{pc} \sqcap \hat{\sigma}^1_{pc} \not\sqsubseteq \sigma$ *or* $\hat{\sigma}^0_{pc} = \hat{\sigma}^1_{pc} \sqsubseteq \sigma$, $r^0_{scope} = r^1_{scope}$, *and* $e_0 = e_1$.

Upgrade Variable Level

$$\frac{\begin{array}{c}\langle \mu, r, x\rangle\ \mathcal{R}_{Scope}\ r_x \qquad \sigma_{pc} \sqsubseteq \Sigma.\mathsf{val}(r_x \cdot x)\\ \Sigma'_{props} = \Sigma.\mathsf{val}[r_x \cdot x \mapsto \sigma \sqcup \Sigma.\mathsf{val}(r_x \cdot x)]\\ \Sigma' = \langle \Sigma.\mathsf{obj}, \Sigma'_{props}, \Sigma.\mathsf{exist}, \Sigma.\mathsf{struct}\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{upg\ var}\ x\ \sigma, \Sigma\rangle \Downarrow_{IF} \langle \mu, \mathtt{tt}, \Sigma', \sigma_{pc}\rangle}$$

Upgrade Property-Value Level

$$\frac{\begin{array}{c}\langle \mu, r, o\rangle\ \mathcal{R}_{Scope}\ r_o \qquad p \in dom(\mu(r_o))\\ \sigma_{pc} \sqsubseteq \Sigma.\mathsf{val}(r_o \cdot p)\\ \Sigma'_{props} = \Sigma.\mathsf{val}[r_o \cdot p \mapsto \sigma \sqcup \Sigma.\mathsf{val}(r_o \cdot p)]\\ \Sigma' = \langle \Sigma.\mathsf{obj}, \Sigma'_{props}, \Sigma.\mathsf{exist}, \Sigma.\mathsf{struct}\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{upg\ prop.val}\ o.p\ \sigma, \Sigma\rangle \Downarrow_{IF} \langle \mu, \mathtt{tt}, \Sigma', \sigma_{pc}\rangle}$$

Upgrade Property-Existence Level

$$\frac{\begin{array}{c}\langle \mu, r, o\rangle\ \mathcal{R}_{Scope}\ r_o \qquad p \in dom(\mu(r_o))\\ \sigma_{pc} \sqsubseteq \Sigma.\mathsf{exist}(r_o \cdot p) \qquad \sigma \sqsubseteq \Sigma.\mathsf{val}(r_o \cdot p) \sqcap \Sigma.\mathsf{struct}(r_o)\\ \Sigma'_{exist} = \Sigma.\mathsf{exist}[r_o \cdot p \mapsto \sigma \sqcup \Sigma.\mathsf{exist}(r_o \cdot p)]\\ \Sigma' = \langle \Sigma.\mathsf{obj}, \Sigma.\mathsf{val}, \Sigma'_{exist}, \Sigma.\mathsf{struct}\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{upg\ prop.exist}\ o.p\ \sigma, \Sigma\rangle \Downarrow_{IF} \langle \mu, \mathtt{tt}, \Sigma', \sigma_{pc}\rangle}$$

Upgrade Structure Level

$$\frac{\begin{array}{c}\langle \mu, r, o\rangle\ \mathcal{R}_{Scope}\ r_o \qquad \sigma_{pc} \sqsubseteq \Sigma.\mathsf{struct}(r_o)\\ \Sigma'_{struct} = \Sigma.\mathsf{struct}\ [r_o \mapsto \sigma \sqcup \Sigma.\mathsf{struct}(r_o)]\\ \Sigma' = \langle \Sigma.\mathsf{obj}, \Sigma.\mathsf{val}, \Sigma.\mathsf{exist}, \Sigma'_{struct}\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{upg\ struct}\ o\ \sigma, \Sigma\rangle \Downarrow_{IF} \langle \mu, \mathtt{tt}, \Sigma', \sigma_{pc}\rangle}$$

Figure 4.4: Semantics of Upgrading Instructions

4.1.6 Labelling Resources at Runtime

Since security labellings are constructed at runtime, it is useful for the programmer to dynamically interact with the current runtime labelling. To this end, following previous approaches in the literature [Hedin 2012, Birgisson 2012], we extend Core JavaScript with the four following language constructs:

- **Variable Upgrade:** upg var x σ — Upgrades the value level of variable x to the *lub* between its current level and σ.
- **Property-Value Upgrade:** upg prop.val $o.p$ σ — Upgrades the *value level* of the property p of the object pointed to by the reference bound to o to the *lub* between its current level and σ.
- **Property-Existence Upgrade:** upg prop.exist $o.p$ σ — Upgrades the *existence level* of the property p of the object pointed to by the reference bound to o to the *lub* between its current level and σ.
- **Structure Upgrade:** upg struct o σ — Upgrades the *structure security level* of the of the object pointed to by the reference bound to o to the *lub* between its current level and σ.

It is important to note that in the cases of the *property-value upgrade* and the *property-existence upgrade*, the variable o is supposed to point to an object that defines a property named p. In other words, the *upgrade instructions do not inspect the prototype-chain.* This has the double advantage of making the semantic rules simpler and of requiring from the the programmer a stricter control over the resources which are to be upgraded. The semantics of the upgrading instructions is given in Figure 4.4.

When using an upgrading instruction, the programmer changes the observable part of the current program state. Therefore, in order to comply with the *no-sensitive-upgrade* strategy, the monitor does not allow visible resources to be upgraded inside invisible contexts. Hence, all four types of upgrades include a constraint meant to prevent sensitive-upgrades. Besides this constraint the Rule [Upgrade Property-Existence Level] includes an additional constraint to guarantee that the existence level of a property p of an object o is always lower than or equal to the structure security level of o and the value level of p.

By inserting upgrade instructions in specific points of a program, the programmer can avoid the runtime errors raised by the monitor when detecting sensitive upgrades. In order to illustrate how to use upgrading instructions to avoid this type of errors, we add the appropriate upgrades to the programs given in Figure 4.4 so that their executions are not aborted by the monitor. The upgrades are depicted in the same colour as the expression that requires their presence.

4.2 Monitor-Inlining

This section presents an information flow monitor-inlining compiler for Core JavaScript, which instruments programs in order to simulate their execution in the monitored semantics presented in Section 4.1.

Type I	**Type II**	**Type III**
$l_{aux} = \mathtt{tt}$, $l = \mathtt{tt}$, upg var l_{aux} H, upg var l H, h ? $(l_{aux} = \mathtt{ff})$, l_{aux} ? $(l = \mathtt{ff})$	$o = \{\}^L$, $o.p = \mathtt{tt}$, $l = \mathtt{tt}$, upg prop.val $o.p$ H, upg var l H, h ? $(o.p = \mathtt{ff})$, $o.p$? $(l = \mathtt{ff})$	$o = \{\}^H$, $o.p = \mathtt{tt}$, $l = \mathtt{tt}$, upg prop.exist $o.p$ H, upg var l H, h ? (delete $o.p$), "p" in o ? $(l = \mathtt{ff})$
Type IV	**Type V**	**Type VI**
$o_h = \{\}^L$, $o_l = \{\}^L$, $l = \mathtt{tt}$, $o_l.p = \mathtt{ff}$, h ? $(o_h = o_l)$, upg prop.val $o.p$ H, upg var l H, $o_h.p = \mathtt{tt}$, $!o_l.p$? $(l = \mathtt{ff})$	$l = \mathtt{tt}$, $o = \{\}^H$, $o.q = \mathtt{ff}$, $prop_h =$ "p", h ? $(prop_h =$ "q"$)$, upg prop.val $o.p$ H, upg prop.exist $o.q$ H, upg var l H, $o[prop_h] = \mathtt{tt}$, $!o.q$? $(l = \mathtt{ff})$	$o_h = \{\}^H$, $o_l = \{\}^H$, $o_h.p = \mathtt{tt}$, $o_l.p = \mathtt{tt}$, $l = \mathtt{tt}$, h ? $o_h = o_l$, upg prop.exist $o_l.p$ H, upg prop.exist $o_h.p$ H, delete $o_h.p$, upg var l H, "p" in o_l ? $(l = \mathtt{ff})$

Table 4.4: Naive Approach vs No-sensitive-upgrade

This instrumentation rests on a technique that consists in pairing up each variable with a new one called its *shadow* variable [Magazinius 2012, Chudnov 2010] that holds its corresponding security level and each property with two *shadow* properties that hold its property-value level and its property-existence level. Since the compiled program has to handle security levels, we include them in the set of program values, which means adding them to the syntax of the language as such, as well as adding two new binary operators corresponding to the order relation ($\sqsubseteq$) and the least upper bound ($\sqcup$).

In the design of the compiler, we assume the existence of a given a set of *internal* variable and property names, denoted by $\mathcal{I}_C$, that do not overlap with those available for the programmer. In particular, the compilation of every *indexed expression* requires extra variables intended to bookkeep the value to which it evaluates and its reading effect, which are later used in the compilation of the expressions that include it. Hence, we assume the set of compiler variables to include two indexed sets of variables $\{\$v_i\}_{i \in \mathbf{N}}$ and $\{\$l_i\}_{i \in \mathbf{N}}$ used to store the levels and the values of intermediate expressions, respectively.

For each variable x the compiler adds a new *shadow* variable, $\$l_x$, that holds its corresponding security level and for each property p the compiler adds two new properties, $\$l_p$ and $\$\overline{l}_p$, that hold its corresponding *value level* and *existence level*. In contrast to variables, whose names are available at compile time, property names can be dynamically computed. Therefore, we assume the existence of two runtime functions, $\$shadow$ and $\$\overline{shadow}$, that given a property name output the name of the shadow properties that hold its value level and existence level, respectively.

Given an expression e to compile, the compiler guarantees that e does not use variable and property names in $\mathcal{I}_C$ by **(1)** statically verifying that the variables in e do not overlap with $\mathcal{I}_C$ and **(2)** dynamically verifying that e does not look-up, create, update, or delete properties whose names belong to $\mathcal{I}_C$. To this end, the compiler makes use of a runtime function $\$legal$ that returns $\mathtt{tt}$ when its argument does not belong to $\mathcal{I}_C$. For clarity, all identifiers reserved for the compiler are prefixed with a dollar sign,

\$. By making sure that compiler identifiers do not overlap with those of the programs to compile, we guarantee the soundness of the proposed transformation even when it receives as input *malicious programs*. Malicious programs try to bypass the inlined runtime enforcement mechanism by rewriting some of its internal variables/properties. For instance, the compilation of the expression $\$l_h = L,\ l = h$ fails, as this program tries to tamper with the internal state of the runtime enforcement mechanism in order to be allowed to leak confidential information. Concretely, this program tries to transfer the content of h to l without raising the level of l by first setting the level of h to *low*.

Besides adding to every object o two additional shadow properties $\$l_p$ and $\$\bar{l}_p$ for every property p in its domain, the inlined monitoring code also adds to o a special property $\$struct$ that stores its structure security level. Hence, given an object $o = [p \mapsto v_0, q \mapsto v_1]$ pointed to by r_o and a labeling Σ, such that: **(1)** $\Sigma.\mathsf{val} = [p \mapsto H, q \mapsto L]$, **(2)** $\Sigma.\mathsf{exist} = [p \mapsto L, q \mapsto L]$, and **(3)** $\Sigma.\mathsf{struct}(r_o) = L$, the instrumented counterpart of o labeled by Σ is:

$$\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$\bar{l}_p \mapsto H, \$\bar{l}_q \mapsto L, \$struct \mapsto L]$$

4.2.1 Formal Specification

The inlining compiler is defined as a function $\mathcal{C}$, given in Figure 4.5 and 4.6. It expects as input an expression e and produces a pair $\langle \hat{e} \mid i \rangle$, where $\hat{e}$ is the expression that simulates the execution of e in the monitored semantics and i an index such that, after the execution of $\hat{e}$, $\$v_i$ stores the value to which e evaluates and $\$l_i$ its corresponding reading effect. Besides the runtime functions $\$shadow$, $\$\overline{shadow}$, and $\$legal$, the compiler makes use of:

- a runtime function $\$check$ that diverges when its argument is different from `tt`;
- a runtime function $\$inspect$ that expects as input an object and a property and outputs the level associated with the corresponding prototype-chain inspection procedure;
- an additional binary operator hasOwnProp that checks whether the object given as its left operand defines the property given as its right one.

In JavaScript, the operator hasOwnProp does not exist; instead, there is a method *hasOwnProperty*, which is accessible to every object *via* its corresponding prototype chain, that checks whether the object on which it is invoked defines the property whose name it receives as input. We chose not to model this feature of the language exactly as it is in the specification in order to keep the model as simple as possible. Doing it otherwise would imply cluttering the already complex semantics of Core JavaScript by having an alternative case for the Rule [Method Call], which would model the semantics of the *hasOwnProperty* method call.

During the evaluation of the instrumented code, the level of the execution context, σ_{pc}, is assumed to be stored in a variable $\$pc$. To this end, function literals are instrumented in order to receive as input the level of the argument and the level of the context in which they are invoked. Function/method calls are instrumented accordingly. Furthermore, the instrumented code of a function/method call must have access to both the return value of the original function/method and the level that is to be associated with that value. Therefore, every function literal returns an object that defines two properties: **(1)** a property $\$v$ that stores the return value of the original function and **(2)** a property $\$l$ that stores the level to be associated with that value.

Each compiler rule precisely mimics the corresponding monitor rule. As done in the presentation of the monitor, constraints are depicted in red and labelling updates are depicted in orange. The compiled code must bookkeep the level and value of indexed expressions. To this end, given an expression e with index i, the compilation of e assigns the value to which it evaluates to a new variable $\$v_i$ and its reading effect to a new variable $\$l_i$. We use light grey for depicting bookkeeping instructions. The compilation of every variable/property assignment and sequence expression does not introduce additional variables because the corresponding value and reading effect are already available in the indexed variables introduced by the corresponding subexpressions.

4.2.2 Correctness

Definition 4.3 presents a *similarity relation* between labelled memories in the monitored semantics and instrumented memories in the original semantics, denoted by $\mathcal{S}$. This relation requires that for every

Value

$$\frac{\hat{e} = \left\{ \begin{array}{l} \$l_i = \$pc, \\ \$v_i = v \end{array} \right.}{\mathcal{C}\langle v^i \rangle = \langle \hat{e} \mid i \rangle}$$

Variable

$$\frac{x \notin \mathcal{I}_C \qquad \hat{e} = \left\{ \begin{array}{l} \$l_i = \$pc \sqcup \$l_x, \\ \$v_i = x \end{array} \right.}{\mathcal{C}\langle x^i \rangle = \langle \hat{e} \mid i \rangle}$$

This

$$\frac{\hat{e} = \left\{ \begin{array}{l} \$l_i = \$pc, \\ \$v_i = \mathsf{this} \end{array} \right.}{\mathcal{C}\langle \mathsf{this}^i \rangle = \langle \hat{e} \mid i \rangle}$$

Binary Operation

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}_0, \\ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k, \\ \$v_i = \$v_i = \$v_j \ \mathsf{op}\ \$v_k \end{array} \right.}{\mathcal{C}\langle e_0 \ \mathsf{op}^i \ e_1 \rangle = \langle \hat{e} \mid i \rangle}$$

Variable Assignment

$$\frac{x \notin \mathcal{I}_C \qquad \langle e' \mid i \rangle = \mathcal{C}\langle e \rangle \qquad \hat{e} = \left\{ \begin{array}{l} e' \\ \$check(\$pc \sqsubseteq \$l_x), \\ \$l_x = \$l_i, \\ x = \$v_i \end{array} \right.}{\mathcal{C}\langle x = e \rangle = \langle e', \hat{e} \mid i \rangle}$$

Property Look-up

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}_0, \\ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k \sqcup \$inspect(\$v_j, \$v_k), \\ (\$v_k \ \mathsf{in}\ \$v_j) \ ? \\ \quad (\$l_i = \$l_i \sqcup \$v_j[\$shadow(\$v_k)]\,), \\ \$check(\$legal(\$v_j))\,, \\ \$v_i = \$v_k[\$v_j] \end{array} \right.}{\mathcal{C}\langle e_0[e_1]^i \rangle = \langle \hat{e} \mid i \rangle}$$

Property Assignment

$$\frac{\langle \hat{e}_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_2 \mid k \rangle = \mathcal{C}\langle e_2 \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}_0,\ \hat{e}_1,\ \hat{e}_2,\ \$check(\$legal(\$v_j)), \\ (\$v_i \ \mathsf{hasOwnProp}\ \$v_j) \ ? \\ \quad \big(\$check(\$l_i \sqcup \$l_j \sqsubseteq \$v_i[\$shadow(\$v_j)])\,\big) \\ \quad : \ \big(\$check(\$l_i \sqcup \$l_j \sqsubseteq \$v_i.\$struct)\,, \\ \qquad \$v_i[\$shadow(\$v_j)] = \$l_i \sqcup \$l_j\,\big), \\ \$v_i[\$shadow(\$v_j)] = \$l_i \sqcup \$l_j \sqcup \$l_k, \\ \$v_i[\$v_j] = \$v_k \end{array} \right.}{\mathcal{C}\langle e_0[e_1] = e_2 \rangle = \langle \hat{e} \mid k \rangle}$$

In Expression

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}_0,\ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k \sqcup \$inspect(\$v_k, \$v_j), \\ (\$v_j \ \mathsf{in}\ \$v_k) \ ? \\ \quad (\$l_i = \$l_i \sqcup \$v_k[\$\overline{shadow}(\$v_j)]\,), \\ \$check(\$legal(\$v_j))\,, \\ \$v_i = \$v_j \ \mathsf{in}\ \$v_k \end{array} \right.}{\mathcal{C}\langle e_0 \ \mathsf{in}^i \ e_1 \rangle = \langle \hat{e} \mid i \rangle}$$

Property Deletion

$$\frac{\langle \hat{e}' \mid j \rangle = \mathcal{C}\langle e \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}' \\ \$check(\$l_j \sqsubseteq \$v_j[\$\bar{l}_p]), \\ \mathsf{delete}\ \$v_j.\$\bar{l}_p, \\ \mathsf{delete}\ \$v_j.\$l_p, \\ \$l_i = \$pc, \\ \$v_i = \mathsf{delete}\ \$v_j.p \end{array} \right.}{\mathcal{C}\langle \mathsf{delete}^i \ e.p \rangle = \langle \hat{e} \mid i \rangle}$$

Object Literal

$$\frac{\hat{e} = \left\{ \begin{array}{l} \$v_i = \{\}, \\ \$v_i.\$struct = \sigma_s, \\ \$v_i.\$l_{proto} = \$pc, \\ \$v_i.\$\bar{l}_{proto} = \$pc, \\ \$l_i = \$pc, \\ \$v_i \end{array} \right.}{\mathcal{C}\langle \{\}^{i,\sigma_s} \rangle = \langle \hat{e} \mid i \rangle}$$

Conditional

$$\frac{\langle \hat{e}_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_2 \mid k \rangle = \mathcal{C}\langle e_2 \rangle \qquad \hat{e} = \left\{ \begin{array}{l} \hat{e}_0,\ \$l_s = \$pc,\ \$pc = \$pc \sqcup \$l_i, \\ \$v_i \ ? \\ \quad \big(\hat{e}_1, \$v_t = \$v_j,\ \$l_t = \$l_j\,\big) \\ \quad : \ \big(\hat{e}_2, \$v_t = \$v_k,\ \$l_t = \$l_k\,\big), \\ \$pc = \$l_s,\ \$v_t \end{array} \right.}{\mathcal{C}\langle e_0 \ ?^{s,t} \ (e_1) : (e_2)\,\rangle = \langle \hat{e} \mid t \rangle}$$

Sequence

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

Figure 4.5: Monitor-Inlining Compiler - Imperative Fragment

Function Literal

$$\frac{\langle \hat{e}_f \mid j\rangle = \mathcal{C}\langle e\rangle \qquad \{i_1,\cdots,i_k\} = indexes(e) \qquad e_{fun} = \left\{\begin{array}{l} \mathsf{function}\,(x, \$l_x, \$pc)\,\{ \\ \quad \mathsf{var}\ y_1, \$l_{y_1}, \cdots, y_n, \$l_{y_n}; \\ \quad \mathsf{var}\ \$v_{i_1}, \$l_{i_1}, \cdots, \$v_{i_k}, \$l_{i_k}; \\ \quad \hat{e}_f, \\ \quad \$ret = \{\}, \\ \quad \$ret.\$v = \$v_j, \\ \quad \$ret.\$l = \$l_j, \\ \quad \$ret \\ \} \end{array}\right. \qquad \hat{e} = \left\{\begin{array}{l} \$v_i = e_{fun}, \\ \$v_i.\$l_{@fscope} = \$pc, \\ \$v_i.\$struct = \$pc, \\ \$l_i = \$pc, \\ \$v_i \end{array}\right.}{\mathcal{C}\langle \mathsf{function}^i(x)\{\mathsf{var}\ y_1,\cdots,y_n;\ e\}\rangle = \langle \hat{e} \mid i\rangle}$$

Function Call

$$\frac{\langle \hat{e}_0 \mid j\rangle = \mathcal{C}\langle e_0\rangle \qquad \langle \hat{e}_1 \mid k\rangle = \mathcal{C}\langle e_1\rangle \qquad \hat{e} = \left\{\begin{array}{l} \hat{e}_0, \\ \hat{e}_1, \\ \$\hat{l}_{ctx} = \$v_j.\$l_{@fscope} \sqcup \$l_j, \\ \$ret = \$v_j(\$v_k, \$l_k \sqcup \$\hat{l}_{ctx}, \$\hat{l}_{ctx}), \\ \$l_i = \$ret.\$l, \\ \$v_i = \$ret.\$v \end{array}\right.}{\mathcal{C}\langle e_0(e_1)^i\rangle = \langle \hat{e} \mid i\rangle}$$

Method Call

$$\frac{\langle \hat{e}_0 \mid j\rangle = \mathcal{C}\langle e_0\rangle \qquad \langle \hat{e}_1 \mid k\rangle = \mathcal{C}\langle e_1\rangle \qquad \langle \hat{e}_2 \mid l\rangle = \mathcal{C}\langle e_2\rangle \qquad \hat{e} = \left\{\begin{array}{l} \hat{e}_0,\ \hat{e}_1,\ \hat{e}_2, \\ \$\hat{l}_{ctx} = \$l_j \sqcup \$l_k \sqcup \$inspect(\$v_k, \$v_j), \\ \$\hat{l}_{ctx} = \$\hat{l}_{ctx} \sqcup \$v_j[\$v_k].\$l_{@fscope}, \\ \$ret = \$v_j[\$v_k](\$v_l, \$\hat{l}_{ctx} \sqcup \$l_l, \$\hat{l}_{ctx}), \\ \$l_i = \$ret.\$l, \\ \$v_i = \$ret.\$v \end{array}\right.}{\mathcal{C}\langle e_0[e_1](e_2)^i\rangle = \langle \hat{e} \mid i\rangle}$$

Figure 4.6: Monitor-Inlining Compiler - Functional Fragment

object in the labelled memory, the corresponding labelling coincide with the instrumented labelling (except for some internal properties whose levels can be automatically inferred) and that the property values of the original object coincide with those of its instrumented counterpart.

Definition 4.3 (Memory Similarity)**.** *A memory μ labeled by Σ is similar to a memory μ', written $\mu, \Sigma\ \mathcal{S}\ \mu'$, if and only if for every reference $r \in dom(\mu)$:*

- $\forall_{p \in dom(o)}\ \mu(r \cdot p) = \mu'(r \cdot p)$;
- $\forall_{p \in dom(o) \backslash \{@scope, @this, @code\}}\ \Sigma.\mathsf{val}(r \cdot p) = \mu'(r \cdot \$l_p)$;
- *If $\mu(r)$ is* **not** *a scope object, then:* $\forall_{p \in dom(o)}\ \Sigma.\mathsf{exist}(r \cdot p) = \mu'(r \cdot \$\bar{l}_p)$;
- *If $\mu(r)$ is* **not** *a scope object, then:* $\Sigma.\mathsf{struct}(r) = \mu'(r \cdot \$struct)$.

The Correctness Theorem states that, provided that a program and its compiled counterpart are evaluated in similar configurations, the evaluation of the original one in the monitored semantics terminates *if and only if* the evaluation of its compilation also terminates in the original semantics, in which case the final memories are similar and the computed values coincide. Therefore, since the monitored semantics only allows secure executions to go through, we guarantee that, when using the inlining compiler, programs are rewritten in such a way that only their secure executions are allowed to terminate.

Theorem 4.3 (Correctness)**.** *Provided that e does not use identifiers in $\mathcal{I}_C$, for any labeled and instrumented configurations $\langle \mu, e, \Sigma\rangle$ and $\langle \mu', e'\rangle$, reference r in $dom(\mu)$, such that $\mu, \Sigma\ \mathcal{S}\ \mu'$ and $\mathcal{C}\langle e\rangle = \langle e' \mid i\rangle$, for some index i; there exists $\langle \mu_f, v, \Sigma_f, \sigma\rangle$ such that $r, \perp\ \vdash \langle \mu, e, \Sigma\rangle \Downarrow_{IF} \langle \mu_f, v, \Sigma, \sigma\rangle$ iff there exists $\langle \mu'_f, v'\rangle$ such that $r \vdash \langle \mu', e'\rangle \Downarrow \langle \mu'_f, v'\rangle$, in which case the following statements hold:* **(1)** $\mu_f, \Sigma_f\ \mathcal{S}\ \mu'_f$, **(2)** $v = v'$, *and* **(3)** $\sigma = \mu'_f(r \cdot \$l_i)$.

4.2.2.1 Establishing Correctness

In order to prove correctness, one must be able to relate the outcome of applying the prototype-chain and the scope-chain look-up procedures in similar memories. To this end, we introduce Lemmas 4.11

and 4.12. Lemma 4.11 states that the results of applying the scope-chain look-up procedure in two similar memories coincide, while Lemma 4.12 states the same but for the prototype-chain look-up procedure.

Lemma 4.11 (Scope-Chain Similarity). *Given two memories μ and μ' and a labeling Σ such that $\mu, \Sigma \ \mathcal{S} \ \mu'$; then, for any reference $r \in \mu$ and identifier x, $\langle \mu, r, x \rangle \ \mathcal{R}_{Scope} \ r_x$ iff $\langle \mu', r, x \rangle \ \mathcal{R}_{Scope} \ r_x$.*

Lemma 4.12 (Prototype-Chain Indistinguishability). *Given two memories μ and μ' and a labeling Σ such that $\mu, \Sigma \ \mathcal{S} \ \mu'$; then, for any two references reference $r, r' \in dom(\mu)$, property p, and security level σ, $\langle \mu, r, p, \Sigma \rangle \ \mathcal{R}_{Proto} \ \langle r', \sigma \rangle$ iff $\langle \mu', r, p \rangle \ \mathcal{R}_{Proto} \ r'$.*

The following two lemmas state two important properties concerning the prototype-chain and the scope-chain inspection procedures that instrumented memories always verify. Lemma 4.14 establishes that the scope object that defines a given variable in a scope-chain coincides is also the scope object that defines its corresponding shadow variable. Analogously, Lemma 4.14 establishes that the object that defines a given property in a prototype-chain is also the object that defines its two corresponding shadow properties.

Lemma 4.13 (Well-Instrumented Scope-Chain). *For any instrumented memory μ, two references r and r_x, and variable x, it holds that: $\langle \mu, r, x \rangle \ \mathcal{R}_{Scope} \ r_x$ iff $\langle \mu, r, \$l_x \rangle \ \mathcal{R}_{Scope} \ r_x$.*

Lemma 4.14 (Well-Instrumented Prototype-Chain). *For any instrumented memory μ, two references r and r_p, and property name p, it holds that: $\langle \mu, r, p \rangle \ \mathcal{R}_{Proto} \ r_p$ iff $\langle \mu, r, \$l_p \rangle \ \mathcal{R}_{Proto} \ r_p$ iff $\langle \mu, r, \$\bar{l}_p \rangle \ \mathcal{R}_{Proto} \ r_p$.*

Finally, Lemma 4.15 states that the value of a bookkeeping variable whose index does not belong to the indexes of the program to compile is not changed by the execution of its respective compilation. In other words, the execution of a compiled program only updates values of bookkeeping variables whose indexes belong to the set of indexes of its original counterpart.

Lemma 4.15 (Well-Instrumented Prototype-Chain). *For any two instrumented memories μ and μ', scope reference r, expression e, indexes i and j, and value value v, such that $\mathcal{C}\langle e \rangle = \langle \hat{e} \mid j \rangle$, $i \notin indexes(e)$, $\$v_i, \$l_i \in dom(\mu(r))$, and $r \vdash \langle \mu, \hat{e} \rangle \Downarrow \langle \mu', v \rangle$, it holds that: $\mu(r \cdot \$v_i) = \mu'(r \cdot \$v_i)$ and $\mu(r \cdot \$l_i) = \mu'(r \cdot \$l_i)$.*

4.3 Related Work

4.3.1 Monitoring Secure Information Flow

Information flow monitors can be divided in two main classes. *Purely dynamic* monitors (such as [Austin 2009] and [Austin 2010]) do not make use of any kind of static analysis. By this, we also mean that purely dynamic monitors do not expect the program to be annotated with the output of a static analysis to be later used at runtime. On the contrary, *hybrid monitors* (such as [Russo 2010]) make use of static analyses to reason about the implicit flows that can arise due to untaken execution paths. Such static analyses are not meant to be performed at runtime. Instead, programs are usually annotated with the output of the analysis, which is then used by the monitor during execution. Austin and Flanagan propose two different strategies for designing purely dynamic information flow monitors. The *no-sensitive-upgrade* strategy [Austin 2009] forbids programs to update the value of low resources in high contexts. Alternatively, the *permissive-upgrade* strategy allows programs to perform sensitive upgrades, but it marks resources that were subject to such upgrades and forbids programs to branch depending on the content of those resources. Our choice for the inlining of a purely dynamic monitor has to do with the fact that the dynamic features of JavaScript make it very difficult to approximate the resources created/updated in untaken program branches. Hedin and Sabelfeld [Hedin 2012] have been the first to design, prove sound, and implement an information flow monitor for a realistic core of JavaScript. Their monitor is purely dynamic and enforces the no-sensitive-upgrade discipline. This monitor has been designed in order to guide a browser instrumentation and not an inlining transformation. Furthermore, it differs from ours in that it labels values instead of variables/properties. Bichhawat et al. [Bichhawat 2014] have recently proposed a hybrid monitor that makes use of a sophisticated static analysis to minimize performance overhead.

4.3.2 Monitor-Inlining Transformations

Chudnov and Naumann [Chudnov 2010] proposed an information flow monitor inlining transformation for a WHILE language, which inlines the hybrid information flow monitor presented in [Russo 2010]. Hence, their inlining compiler includes a simple static analysis that estimates the set of variables updated in untaken program branches. Simultaneously, Magazinius et al. [Magazinius 2012] propose the inlining of a purely dynamic information flow monitor that enforces the no-sensitive-upgrade discipline for a simple imperative language that features global functions, a let construct, and an *eval* expression that allows for dynamic code evaluation. Both compilers pair up each variable with a *shadow* variable. We extend this technique to handle object properties by pairing up each property with two shadow properties. The languages modeled in both [Chudnov 2010] and [Magazinius 2012] only feature primitive values and do not feature scope composition (in [Chudnov 2010] there are no functions and in [Magazinius 2012] every function is executed in a "clean" environment and does not produce side-effects). Hence, in both [Chudnov 2010] and [Magazinius 2012], the reading effect of an expression e corresponds to the least upper bound on the levels of the variables of e. Therefore, the instrumented code for computing the level of e is simply $\$l_{x_1} \sqcup \cdots \sqcup \$l_{x_n}$, where $\{x_1, \cdots, x_n\}$ are the variables that explicitly occur in e. In Core JavaScript (as in JavaScript) this does not hold. First, one can immediately see that expressions that feature property look-ups or function/method calls do not generally verify this property. Second, expressions may be composed of expressions that have side effects. Therefore, the level associated with the whole expression can actually be lower than the least upper bound on the levels of the variables that it includes. As an example, consider the expression $(x = y) + x$. Since $x = y$ evaluates to the value of y (besides assigning the value of y to x), the level of the whole expression only depends on the initial level of y. In order to handle these two issues, the inlining transformation must introduce extra variables to keep track of the values and levels of intermediate expressions. Finally, both [Chudnov 2010] and [Magazinius 2012] ignore the problem of malicious programs.

Chapter 5

From Static to Hybrid Information Flow Control in Core JavaScript

Contents

One of the major issues in developing static analyses for JavaScript is the fact that "property names can be computed using string operations" [Maffeis 2009], which renders intractable the problem of deciding at the static level which property is actually being accessed in a given property look-up. Consider the following program:

```
o = {}, o.secret_prop = secret_input(),
o.public_prop = public_input(), public_out = o[f()]
```

that creates an object `o` with two properties `secret_prop` and `public_prop` which are respectively assigned to a secret input and a public input (read via functions `secret_input` and `public_input`) and then assigns one of them to a public output depending on the return value of function `f`. In this example, deciding which property is assigned to the public output is equivalent to predicting the dynamic behavior of function `f`, which is, in general, undecidable. In order to overcome this issue, previous analyses for enforcing confinement properties in JavaScript (such as that of [Maffeis 2009]) have chosen to restrict the targeted language subset, excluding property look-ups with arbitrary expressions.

We propose a new approach (Section 5.4), exploiting the connections between static and runtime analysis to avoid rejecting programs that are in fact secure. The key insight of our approach is that, since we aim at enforcing **termination insensitive** noninterference, the analysis may infer a set of assertions under which a program can be securely accepted and then dynamically verify whether or not these assertions hold. The original program is instrumented in such a way that if the assertions under which it is *conditionally accepted* fail to hold, its instrumentation diverges. For instance, the example presented above cannot be statically considered secure (for an arbitrary function `f`), since in general it is not possible to decide whether a function produces a given output. However, the following modified version of this program:

```
o = {}, o.secret_prop = secret_input(),
o.public_prop = public_input(), _x = f(),
(_x != "secret_prop") ? public_out = o[f()] : _diverge()
```

can be securely accepted, since it diverges whenever `f` evaluates to `"secret_prop"`. Hence, we guarantee that the potential illegal information flow never occurs.

5.1 Annotating Core JavaScript

In order to ease the specification of the static analysis, we modify the syntax of Core JavaScript so that, (as in as in [Taly 2011]), property look-ups, method calls, and property assignments are annotated with a set P of the properties to which the corresponding expression may evaluate, which we call a *look-up annotation*. For instance, in the expression `o`[`e`, {`"foo"`, `"bar"`, `"baz"`}], the look-up annotation means that `e` always evaluates to a string equal to `"foo"`, `"bar"`, or `"baz"`. We similarly annotate the occurrences of the `in` expression with the set of properties that may be checked. Furthermore, object literals as well as the variables declared in the body of a function are annotated with their respective security types (which are explained later in this chapter). The modified syntax is given below:

$$\begin{array}{lcll}
e & ::= & \ldots & \\
 & & \mid\ \mathsf{function}^{\dot\tau,i}(x)\{\mathsf{var}^{\dot\tau_1,\cdots,\dot\tau_n}\ y_1,\cdots,y_n;\ e\} & \text{function literal} \\
 & & \mid\ \{\}^{\dot\tau,i} & \text{object literal} \\
 & & \mid\ e_0[e_1,P](e_2)^i & \text{method call} \\
 & & \mid\ e_0[e_1,P]^i & \text{property look-up} \\
 & & \mid\ e_0[e_1,P] = e_2 & \text{property assignment} \\
 & & \mid\ e_0\ \mathsf{in}_i^P\ e_1 & \text{membership testing}
\end{array}$$

We say that a look-up annotation P is *correct* if the expression to which it applies always evaluates to a string in P. Moreover, we say that P is *minimal* if there is no other correct P' such that $P' \subset P$. It is trivial to instrument a program so that it diverges if its look-up annotations are not correct. For instance, one could easily modify the specification of the hybrid type system to ensure the correctness of look-up annotations. This would, however, clutter up the presentation. Hence, we leave it implicit and in the rest of this chapter assume that look-up annotations are correct. But they do not have to be minimal – the look-up annotation corresponding to the set $\mathcal{S}tr$ of all strings is always correct. We say that two expressions e and e' are *equal up to look-up annotations*, written $e \equiv e'$, if they only differ in look-up annotations. Whenever a look-up annotation is omitted, it is assumed to be $\mathcal{S}tr$, and the notation `o.p` is used as an abbreviation for `o`[`"p"`,{`"p"`}].

5.2 Security Types for Core JavaScript

In JavaScript, the programmer can dynamically add and remove properties from objects. In fact, objects are commonly used as tables whose keys are computed at runtime. Hence, in many contexts, it is not realistic to expect the programmer to statically know the properties of the objects that are created at runtime. However, security-wise, the programmer often knows the security level of the contents of an object even when its actual properties are not known. For instance, in the Contact Manager example, the precise structure of `contact_list` cannot be statically known because the last names in its domain are dynamic inputs. Nevertheless, the programmer should be allowed to specify a security policy stating, for example, that the e-mail address of every contact in `contact_list` is confidential and therefore of level H.

In contrast to class-based languages, where method types are specified inside their classes, JavaScript functions are first-class values which can be defined anywhere in the code and later assigned to properties of arbitrary objects. This creates a dependency between types for functions and types for objects, because object types include the types of their methods and function types include the type of the objects to which the keyword `this` is bound during execution. To break this circularity, we make use of equi-recursive types. However, to keep the presentation fairly simple, we restrict the occurrence of type variables to the type of `this` in function types.

Every security type $\dot\tau = \tau^\sigma$ is obtained by pairing up a *raw* type τ with a security level σ, that gives an upper bound on the levels of the resources on which the values of that type may depend. For instance, a primitive value of type PRIM^L may only depend on *low* resources. The same applies to an object o of type $\mu\kappa.\langle p^L : \mathsf{PRIM}^H\rangle^L$. However, the value associated with o's property p may depend on *high* resources. Let p, σ, and κ range over the sets of strings, security levels, and type variables. The syntax of raw types is as follows:

$$\begin{array}{lcl}
\tau & ::= & \mathsf{PRIM} \mid \langle \dot\tau.\dot\tau \xrightarrow{\sigma} \dot\tau\rangle \mid \langle \kappa.\dot\tau \xrightarrow{\sigma} \dot\tau\rangle \\
 & & \mid \mu\kappa.\langle p^\sigma : \dot\tau, \cdots, p^\sigma : \dot\tau, *^\sigma : \dot\tau\rangle \mid \mu\kappa.\langle p^\sigma : \dot\tau, \cdots, p^\sigma : \dot\tau\rangle
\end{array}$$

$$\dot{\tau}_{contact} = \mu\kappa. \left\langle \begin{array}{c} fst^L : \textsf{PRIM}^L, lst^L : \textsf{PRIM}^L, id^L : \textsf{PRIM}^H, printContact^L : \langle \kappa.\textsf{PRIM}^L \xrightarrow{H} \textsf{PRIM}^L \rangle^L, \\ makeFavorite^L : \langle \kappa.\textsf{PRIM}^L \xrightarrow{H} \textsf{PRIM}^L \rangle^L, isFavorite^L : \langle \kappa.\textsf{PRIM}^L \xrightarrow{H} \textsf{PRIM}^H \rangle^L, \\ unFavorite^L : \langle \kappa.\textsf{PRIM}^L \xrightarrow{H} \textsf{PRIM}^H \rangle^L, favorite^H : \textsf{PRIM}^H, \_prot\_^L : \dot{\tau}_{proto\_contact} \end{array} \right\rangle^L$$

$$\dot{\tau}_{CM} = \mu\kappa. \left\langle \begin{array}{c} proto\_contact^L : \dot{\tau}_{contact}, contact\_list^L : \mu\kappa.\langle *^L : \dot{\tau}_{contact}, L\rangle^L, \\ create\_contact^L : \langle \kappa.(\textsf{PRIM}^L, \textsf{PRIM}^L, \textsf{PRIM}^H) \xrightarrow{L} \dot{\tau}_{contact} \rangle^L, \\ store\_contact^L : \langle \kappa.(\dot{\tau}_{contact}, \textsf{PRIM}^L) \xrightarrow{L} \dot{\tau}_{contact} \rangle^L, \_prot\_^L : \textsf{PRIM}^L \end{array} \right\rangle^L$$

Figure 5.1: Typing Environment for the Contact Manager - $\Gamma_{CM} = [CM \mapsto \dot{\tau}_{CM}]$

We denote by $\mathcal{T}$ the set of all security types. Given a security type $\dot{\tau}$, $lev(\dot{\tau})$ denotes its level and $\lfloor \dot{\tau} \rfloor$ its raw type. For instance, $\sigma(\textsf{PRIM}^L) = L$ and $\lfloor \textsf{PRIM}^L \rfloor = \textsf{PRIM}$. We define $\dot{\tau}^\sigma$ as $\lfloor \dot{\tau} \rfloor^{lev(\dot{\tau}) \sqcup \sigma}$. Hence, $(\textsf{PRIM}^L)^H = \textsf{PRIM}^H$. A typing environment Γ is a mapping from variables to types.

The type $\textsf{PRIM}$ is the type of all primitive values. The type $\langle \dot{\tau}_0.\dot{\tau}_1 \xrightarrow{\sigma} \dot{\tau}_2 \rangle$ is the type of all functions that map values of type $\dot{\tau}_1$ to values of type $\dot{\tau}_2$ and during the execution of which the keyword `this` is bound to an object of type $\dot{\tau}_0$. The level σ is the *writing effect* [Sabelfeld 2003a] of the function, i.e., a lower bound on the levels of the resources created/updated during its execution. The type $\mu\kappa.\langle p_0^{\sigma_0} : \dot{\tau}_0, \cdots, p_n^{\sigma_n} : \dot{\tau}_n, *^{\sigma_*} : \dot{\tau}_* \rangle$ is the type of all objects that **potentially** define properties $p_0, \cdots, p_n$, mapping each property p_i to a value of type $\dot{\tau}_i$. The type assigned to the $*$ is the *default type*. Every property p_i is additionally associated with an *existence level* σ_i. The level σ_* is the *default existence level*. We use the notation $dom(\dot{\tau})$ for the set containing the properties that appear in $\dot{\tau}$ (including $*$ if it is present), and the notation $*(\dot{\tau})$ for the pair $(\sigma_*, \dot{\tau}_*)$ consisting of the default existence level and security type of $\dot{\tau}$.

The fact that an object has type $\dot{\tau}$ does not mean that it defines all properties in $dom(\dot{\tau})$, but rather that it **potentially** defines the properties in $dom(\dot{\tau})$. Moreover, if $* \notin dom(\dot{\tau})$, then o is assumed to be *non-extensible*, meaning that only properties in $dom(\dot{\tau})$ can be added to o. This is statically enforced by the type systems presented in this section. Figure 5.1 presents a typing environment for the Contact Manager example. We omit the specification of the type $\dot{\tau}_{proto\_contact}$ that coincides with $\dot{\tau}_{contact}$ in every property except in `_prot_` for which it does not define a mapping, since objects of that type are not supposed to have a prototype.[1]

It is useful to define a function $\Rsh$ that receives as input an object security type $\dot{\tau}$ and a string p and outputs a pair consisting of the existence level and the security type with which $\dot{\tau}$ associates p:

$$\Rsh(\dot{\tau}, p) = \begin{cases} (\sigma_i, \{\dot{\tau}/\kappa\}\dot{\tau}_p) & \text{if } \dot{\tau} = \mu\kappa.\langle \cdots, p^{\sigma_i} : \dot{\tau}_p, \cdots \rangle^\sigma \\ (\sigma_*, \{\dot{\tau}/\kappa\}\dot{\tau}_*) & \text{if } \begin{array}{l} \dot{\tau} = \mu\kappa.\langle \cdots, *^{\sigma_*} : \dot{\tau}_*, \cdots \rangle^\sigma \\ p \notin dom(\dot{\tau}) \end{array} \end{cases}$$

where $\{\dot{\tau}_0/\kappa\}\dot{\tau}_1$ denotes the capture-avoiding substitution of κ for $\dot{\tau}_0$ in $\dot{\tau}_1$. Interestingly, given an object type $\dot{\tau}$, if we define $\Rsh(\dot{\tau}) : \mathcal{S}tr \to \mathcal{T}$, as the function that maps every identifier p to the second element of $\Rsh(\dot{\tau}, p)$, one can interpret an object type as a typing environment. Indeed, programs must be typed in a typing environment matching the type of the *global object* $\dot{\tau}_{glob}$, meaning that: if $\Gamma(x) = \dot{\tau}_x$, then $\Rsh(\dot{\tau}_{glob}, x) = (\sigma', \dot{\tau}_x)$.

5.2.1 Admissible Prototypes

An important aspect of object types is that they must reflect the whole prototype-chain accessible through the corresponding objects. Hence, in the Contact Manager example, the security type assigned to contact objects also includes the methods that the corresponding prototype implements. Since every object type must reflect the whole prototype-chain accessible through the corresponding objects, not all types can be used as the *type of the prototype* for the objects of a given type. Consider, for instance, an object o_0 of type $\dot{\tau}_0 = \mu\kappa.\langle p^L : \textsf{PRIM}^L, \_prot\_^L : \_ \rangle$ and an object o_1 of type $\dot{\tau}_1 = \mu\kappa.\langle p^L : \mu\kappa.\langle *^L : \textsf{PRIM}^L \rangle^L \rangle$. Suppose we set $\dot{\tau}_1$ as the type of the prototype in $\dot{\tau}_0$. Then, the look-up of p in o_0 may yield two different

[1]Note that in real JavaScript every object has an implicit prototype: `Object.prototype`.

types of values (besides `undefined`, if neither o_0 nor o_1 defines p). It yields a value of type $\textsc{prim}^L$ when object o_0 defines p and an object of type $\mu\kappa.\langle *^L : \textsc{prim}^L\rangle^L$ when o_0 does not define p and o_1 defines p. In order to overcome this problem, we restrict what types can be legally used for the prototype of a given object type. We say that $\dot\tau_1$ is a *consistent prototype type* for $\dot\tau_0$ if:

- $\dot\tau_1$ does not define a default type – $* \notin dom(\tau_1)$;
- $\dot\tau_1$ coincides with $\dot\tau_0$ for all properties in its domain – $dom(\tau_1) \subseteq dom(\tau_0)$ and $p \in dom(\tau_1)\backslash\{\_prot\_\}$, $\Rsh(\tau_0, p) = \Rsh(\tau_1, p)$.

5.3 A Type System for Information Flow Control in Core JavaScript

We now present a static type system for securing information flow in Core JavaScript. The rules, presented in Figure 5.2, use typing judgements of the form $\Gamma \vdash e : \dot\tau, \sigma$, where **(1)** Γ is the typing environment, **(2)** e the expression to be typed, **(3)** $\dot\tau$ the type that is assigned to it, and **(4)** σ its *writing effect*, that is, a lower bound on the levels of the resources that are updated/created when e is evaluated.

The type systems presented here assumes two basic restrictions on the syntax of security types. First, we require the existence level of a property to be lower than or equal to the level that annotates its corresponding security type. This restriction forbids the specification of an object type that associates an invisible property with a visible value. Second, we require the security level that annotates an object type to be higher than or equal to the level that annotates the type of its prototype. This constraint is meant to prevent leaks *via* prototype mutations. If the level of the prototype of an object o is *high*, then the prototype of o is allowed to change in a *high* context. However, such changes remain invisible to a *low* observer, because the level of o is itself *high*, meaning that a *low* observer can never see any of the contents of o.

In order to type expressions that either result from the combination of subexpressions with different types, or whose evaluation may yield values of different types (for instance, a property look-up with an imprecise look-up annotation), the type system makes use of an ordering on security types. The ordering $\sqsubseteq$ on security levels induces a simple ordering $\preceq$ on security types: $\dot\tau_0 \preceq \dot\tau_1$ *iff* $lev(\dot\tau_0) \sqsubseteq lev(\dot\tau_1)$ and $\lfloor\dot\tau_0\rfloor \equiv \lfloor\dot\tau_1\rfloor$, where $\equiv$ stands for syntactic equality up to arbitrary unfoldings of raw types [Anderson 2005]. Every two object security types in the subtyping relation need to have the same corresponding raw type, because, while property look-ups are *covariant* with the type of the property, property assignments are *contravariant*. Concretely, given an object of type $\dot\tau_0 = \mu\kappa.\langle p^L : \textsc{prim}^L\rangle^L$ bound to $\mathtt{x}$ and an object of type $\dot\tau_1 = \mu\kappa.\langle p^L : \textsc{prim}^H\rangle^L$ bound to $\mathtt{y}$, if we let $\dot\tau_0 \preceq \dot\tau_1$, the expression $\mathtt{y} = \mathtt{x}$, $\mathtt{y.p} = \mathtt{h}$, which is **not** noninterferent, would be typable. Given a raw type τ, the set $\{\dot\tau \mid \lfloor\dot\tau\rfloor \equiv \tau\}$ of its corresponding security types (ordered by $\preceq$) forms a lattice. The corresponding *lub* and *glb* $\curlyvee, \curlywedge : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ are defined as follows: $\dot\tau_0 \, {}^{\curlyvee}_{\curlywedge} \, \dot\tau_1 = \dot\tau \Leftrightarrow \lfloor\dot\tau\rfloor \equiv \lfloor\dot\tau_0\rfloor \equiv \lfloor\dot\tau_1\rfloor \wedge lev(\dot\tau) = lev(\dot\tau_0) {}^{\sqcup}_{\sqcap} lev(\dot\tau_1)$. Using the notions of *lub* and *glb* between security types, we extend function $\Rsh$ to arbitrary sets of properties in the two following ways:

$$\begin{aligned}
\Rsh_{\uparrow}(\dot\tau, P) &= (\sqcup\{\hat\sigma \mid p \in P \wedge \Rsh(\dot\tau, p) = (\hat\sigma, \dot\tau')\}, \curlyvee\{\dot\tau' \mid p \in P \wedge \Rsh(\dot\tau, p) = (\sigma, \dot\tau')\}) \\
\Rsh_{\downarrow}(\dot\tau, P) &= (\sqcap\{\hat\sigma \mid p \in P \wedge \Rsh(\dot\tau, p) = (\hat\sigma, \dot\tau')\}, \curlywedge\{\dot\tau' \mid p \in P \wedge \Rsh(\dot\tau, p) = (\sigma, \dot\tau')\})
\end{aligned}$$

While $\Rsh_{\uparrow}$ is used for the typing of property look-ups, in expressions, and method calls (which are covariant with the type of the corresponding property), $\Rsh_{\downarrow}$ is used for the typing of property assignments (which are contravariant with the type of the corresponding property).

In the following, we give a brief description of the rules that better illustrate the information flows specific to Core JavaScript and refer to [Sabelfeld 2003a] for a comprehensive presentation of a classical type system for IFC. In the Rule [Property Assignment], the raw type of the property that is being assigned ($\lfloor\dot\tau\rfloor$) must coincide with the raw type of the expression to which it is being assigned ($\lfloor\dot\tau_2\rfloor$). The constraint $lev(\dot\tau) \sqsubseteq lev(\dot\tau_2)$ prevents the *explicit flow* resulting from the assignment of a *high* value to a *low* property, whereas the constraint $lev(\dot\tau_0) \sqcup lev(\dot\tau_1) \sqsubseteq \sigma$ prevents the *implicit flows* – one cannot create a property with a *low* existence level depending on *high* values. Moreover, one cannot update a property associated with a *low* value depending on *high* values. However, the former constraint subsumes the latter. In the Rule [Property Deletion], the security level that annotates the type of the object whose property is being deleted ($lev(\dot\tau)$) must be lower than or equal to the existence level of that property (σ'). The reason is that one cannot delete a visible property depending on secret information. In both

$$\textsc{Val}\quad \Gamma \vdash v : \textsf{PRIM}^{\bot}, \top \qquad \textsc{This}\quad \Gamma \vdash \textsf{this} : \Gamma(\textsf{this}), \top \qquad \textsc{Var}\quad \Gamma \vdash x : \Gamma(x), \top \qquad \textsc{Object Literal}\quad \Gamma \vdash \{\}^{\dot\tau, i} : \dot\tau, lev(\tau)$$

$$\textsc{Binary Operation}\quad \frac{\forall_{i=0,1} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i}{\Gamma \vdash e_0 \;\textsf{op}\; e_1 : \dot\tau_0 \curlyvee \dot\tau_1, \sigma_0 \sqcap \sigma_1}$$

$$\textsc{Variable Assignment}\quad \frac{\begin{array}{c}\Gamma \vdash e : \dot\tau, \sigma \qquad \dot\tau \preceq \Gamma(x)\\ \sigma' = \sigma \sqcap lev(\Gamma(x))\end{array}}{\Gamma \vdash x = e : \dot\tau, \sigma'}$$

$$\textsc{Property Look-up}\quad \frac{\begin{array}{c}\forall_{i=0,1} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i\\ \Rsh_{\uparrow} (\dot\tau_0, P) = (\sigma, \dot\tau) \quad \sigma' = \sigma_0 \sqcap \sigma_1\end{array}}{\Gamma \vdash e_0[e_1, P] : \dot\tau^{lev(\dot\tau_0) \sqcup lev(\dot\tau_1)}, \sigma'}$$

$$\textsc{In Expression}\quad \frac{\begin{array}{c}\forall_{i=0,1} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i \qquad \sigma = \sigma_0 \sqcap \sigma_1\\ \Rsh_{\uparrow} (\dot\tau_1, P) = (\sigma', \dot\tau) \qquad \sigma'' = lev(\dot\tau_0) \sqcup lev(\dot\tau_1)\end{array}}{\Gamma \vdash e_0 \;\textsf{in}^P\; e_1 : \textsf{PRIM}^{\sigma' \sqcup \sigma''}, \sigma}$$

$$\textsc{Property Assignment}\quad \frac{\begin{array}{c}\forall_{i=0,1,2} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i \qquad \Rsh_{\downarrow} (\dot\tau_0, P) = (\sigma, \dot\tau)\\ \dot\tau_2 \preceq \dot\tau \qquad \sigma(\dot\tau_0) \sqcup \sigma(\dot\tau_1) \sqsubseteq \sigma\\ * \notin dom(\dot\tau_0) \Rightarrow P \subseteq dom(\dot\tau_0)\end{array}}{\Gamma \vdash e_0[e_1, P] = e_2 : \dot\tau_2, \sigma_0 \sqcap \sigma_1 \sqcap \sigma_2 \sqcap \sigma}$$

$$\textsc{Function Call}\quad \frac{\begin{array}{c}\Gamma \vdash e_0 : \langle \dot\tau'_0 . \dot\tau'_1 \xrightarrow{\hat\sigma} \dot\tau'_2 \rangle^{\hat\sigma'}, \sigma_0\\ \Gamma \vdash e_1 : \dot\tau_1, \sigma_1 \qquad \dot\tau_{global} \preceq \dot\tau'_0\\ \dot\tau_1 \preceq \dot\tau'_1 \qquad \hat\sigma' \sqsubseteq \hat\sigma\end{array}}{\Gamma \vdash e_0(e_1) : (\dot\tau'_2)^{\hat\sigma'}, \sigma_0 \sqcap \sigma_1 \sqcap \hat\sigma}$$

$$\textsc{Method Call}\quad \frac{\begin{array}{c}\forall_{i=0,1,2} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i \qquad \Rsh_{\uparrow} (\dot\tau_0, P) = (\sigma, \langle \dot\tau'_0 . \dot\tau'_1 \xrightarrow{\hat\sigma} \dot\tau'_2 \rangle^{\hat\sigma'})\\ \sigma' = \hat\sigma' \sqcup \sigma(\dot\tau_0) \sqcap \sigma(\dot\tau_1) \qquad \dot\tau_0 \preceq \dot\tau'_0 \quad \dot\tau_2 \preceq \dot\tau'_1 \quad \sigma' \sqsubseteq \hat\sigma\end{array}}{\Gamma \vdash e_0[e_1, P](e_2) : (\dot\tau'_2)^{\sigma'}, \sigma_0 \sqcap \sigma_1 \sqcap \sigma_2 \sqcap \hat\sigma}$$

$$\textsc{Property Deletion}\quad \frac{\begin{array}{c}\Gamma \vdash e : \dot\tau, \sigma \qquad \Rsh (\dot\tau, p) = (\sigma', \dot\tau')\\ lev(\dot\tau) \sqsubseteq \sigma'\end{array}}{\Gamma \vdash \textsf{delete}\; e.p : \textsf{PRIM}^{\bot}, \sigma \sqcap \sigma'}$$

$$\textsc{Sequence}\quad \frac{\forall_{i=0,1} \cdot \Gamma \vdash e_i : \dot\tau_i, \sigma_i}{\Gamma \vdash e_0, e_1 : \dot\tau_1, \sigma_0 \sqcap \sigma_1}$$

$$\textsc{Conditional Expression}\quad \frac{\begin{array}{c}\forall_{i=0,1,2} \cdot \Gamma \vdash e_i : \dot\tau_i, \sqcup_i \quad \sigma = \sigma_0 \sqcap \sigma_1 \sqcap \sigma_2\\ \sigma(\dot\tau_0) \sqsubseteq \sigma_1 \sqcap \sigma_2\end{array}}{\Gamma \vdash e_0 \;?\; (e_1) : (e_2) : (\dot\tau_1 \curlyvee \dot\tau_2)^{\sigma(\dot\tau_0)}, \sigma}$$

$$\textsc{Function Literal}\quad \frac{\begin{array}{c}\dot\tau = \langle \dot\tau'_0 . \dot\tau'_1 \xrightarrow{\hat\sigma} \dot\tau'_2 \rangle^{\sigma}\\ \Gamma\,[\textsf{this} \mapsto \dot\tau'_0, x \mapsto \dot\tau'_1, y_1 \mapsto \dot\tau_1, \cdots, y_n \mapsto \dot\tau_n] \vdash e : \dot\tau'_2, \hat\sigma\end{array}}{\Gamma \vdash \textsf{function}^{\dot\tau}(x)\{\textsf{var}^{\dot\tau_1, \cdots, \dot\tau_n}\; y_1, \cdots, y_n;\; e\} : \dot\tau, \sigma}$$

Figure 5.2: Typing Secure Information Flow in Core JavaScript

rules, the existence level of the property being assigned/deleted is included in the writing effect of the respective expression in order to prevent the creation/deletion of visible properties in invisible contexts. Finally, the Rule [Method Call] checks whether the types of the object ($\dot\tau_0$) and the argument ($\dot\tau_2$) match the types of the keyword `this` ($\dot\tau'_0$) and the formal parameter ($\dot\tau'_1$) of the method being invoked. The constraint $\sigma' \sqsubseteq \hat\sigma$ prevents the calling of a method that creates/updates *low* memory depending on *high* values. The soundness of the proposed type system is established in Theorem 5.1.

Theorem 5.1 (Noninterference)**.** *For any expression e and typing environment Γ such that $\Gamma \vdash e : \tau, \sigma$, it holds that e is noninterferent w.r.t. Γ.*

5.4 A Hybrid Approach for Information Flow Control in Core JavaScript

The precision of the purely static type system heavily depends on the precision of look-up annotations. For instance, a property look-up is typable only if all properties in the corresponding look-up annotation are associated with the same raw type. In this section, we modify this type system so as to make its precision independent of the precision of look-up annotations. The key insight is that, since our goal is to verify **termination insensitive** noninterference, we can defer failure to execution time. Hence, instead of rejecting a program based on imprecise look-up annotations, the hybrid type system infers a set of assertions under which a program can be securely accepted and instruments it so as to dynamically check whether these assertions hold. The instrumented version diverges if the assertions under which the original version was *conditionally accepted* fail to hold at runtime.

In order to be able to reason about intermediate states of the execution, the type system makes use of an indexed set of variables $\mathcal{I}_C$. These variables are used for bookkeeping the values of intermediate expressions and are not available for the programmer. Since one can easily instrument a program so that it diverges when trying to read/write reserved variables, we can assume that program variables do not overlap with those in $\mathcal{I}_C$. The runtime assertions generated by the type system are described by the following grammar:

$$\omega ::= \$v_i \in V \mid v \in V \mid \texttt{tt} \mid \omega \vee \omega \mid \omega \wedge \omega \mid \neg\omega$$

where $\$v_i$ is the i-th variable of $\mathcal{I}_C$ and V an arbitrary set of primitive values. We consider two types of *elementary assertions.* An elementary assertion $v \in V$ holds if the value v is contained in V. An *elementary assertion* $\$v_i \in V$ holds in a memory μ in the scope-chain starting from r, written $\mu, r \vDash \$v_i \in V$, if $\$v_i$ is bound to a value in V in that scope. Formally, $\langle \mu, r, \$v_i \rangle \ \mathcal{R}_{Scope}\ r'$ and $\mu(r')(\$v_i) \in P$. The remaining assertions are interpretated as in classical propositional logic.

In this section, we use as a running example the program $\mathtt{x[y] = u[v] + z}$, to be typed using the following typing environment:

$$\begin{gathered}
\Gamma(x) = \mu\kappa.\langle p_0^L : \textsf{PRIM}^H, p_1^L : \textsf{PRIM}^L, *^L : \textsf{PRIM}^L \rangle^L \\
\Gamma(u) = \mu\kappa.\langle q_0^L : \textsf{PRIM}^H, q_1^L : \textsf{PRIM}^L, *^L : \textsf{PRIM}^H \rangle^L \\
\Gamma(z) = \Gamma(y) = \Gamma(v) = \textsf{PRIM}^L
\end{gathered}$$

This program is not typable using the static type system, because the left-hand side expression is typed with $\textsf{PRIM}^L$ (since the type system uses $\Rsh_\downarrow$ to determine its type), while the right-hand side expression is typed with $\textsf{PRIM}^H$ (since the type system uses $\Rsh_\uparrow$ to determine its type).[2] However, since the look-up annotations of this program are very imprecise, it can be the case that the potential illegal flows, which cause the static type system to reject it, never actually happen. Hence, instead of assigning a single security type and a single writing effect to each expression, the hybrid type system assigns it a set T of *possible* security types and a set L of *possible* writing effects. Each type $\dot{\tau}$ in T and each security level σ in L is paired up with an assertion ω that describes "when" the expression is correctly typed by $\dot{\tau}$ or has writing effect σ. For instance, the look-up expressions $\mathtt{x[y]}$ and $\mathtt{u[v]}$ are respectively typed with $T_{x[y]} = \{(\textsf{PRIM}^H, \$v_i \in \{p_0\}), (\textsf{PRIM}^L, \$v_i \in \{p_1\}), (\textsf{PRIM}^L, \neg(\$v_i \in \{p_0, p_1\}))\}$ and $T_{u[v]} = \{(\textsf{PRIM}^L, \$v_j \in \{q_1\}), (\textsf{PRIM}^H, \$v_j \in \{q_0\}), (\textsf{PRIM}^H, \neg(\$v_j \in \{q_0, q_1\}))\}$, where $\$v_i$ and $\$v_j$ are the variables of the type system that hold the values to which $\mathtt{y}$ and $\mathtt{v}$ evaluate in that context.

It is useful to define a function $\Rsh^?$ that **expects** as input an object type $\dot{\tau}$, a set P of properties to inspect, and an expression e that evaluates to the actual property being inspected[3] and **generates** a set of triples of the form $(\sigma, \dot{\tau}', \omega)$. Each of these triples consists of a security level σ, a security type $\dot{\tau}'$, and the assertion ω that must hold so that the actual property being looked-up has existence level σ and security type $\dot{\tau}'$. Formally, letting $LT^{\dot{\tau},P,e} = \{(\sigma, \dot{\tau}', (e \in \{p\}) \mid p \in P \cap dom(\dot{\tau}) \wedge \Rsh(\dot{\tau}, p) = (\sigma, \dot{\tau}')\}$ and $LT_*^{\dot{\tau},P,e} = \{(\sigma_*, \tau_*, \neg(e \in dom(\dot{\tau}) \cap P))\}$ (where $*(\dot{\tau}) = (\sigma_*, \tau_*)$), $\Rsh^?$ is defined as follows:

$$\Rsh^?(\dot{\tau}, P, e) = \begin{cases} LT^{\dot{\tau},P,e} & \text{if } P \subseteq dom(\dot{\tau}) \\ LT^{\dot{\tau},P,e} \cup LT_*^{\dot{\tau},P,e} & \text{if } P \not\subseteq dom(\dot{\tau}) \end{cases}$$

We extend $\Rsh^?$ to sets of object security types paired up with runtime assertions in the following way: $\Rsh^?(T, P, e) = \{(\sigma, \dot{\tau}', \omega \wedge \omega') \mid (\dot{\tau}, \omega) \in T \ \wedge \ (\sigma, \dot{\tau}', \omega') \in \Rsh^?(\dot{\tau}, P, e)\}$. Given a set LT of triples of the form $(\sigma, \dot{\tau}, \omega)$, we denote by $\pi_{\texttt{lev}}(LT)$ ($\pi_{\texttt{type}}(LT)$, resp.) the set of pairs obtained from LT by removing from each triple the security type (the security level, resp.). Observe that $\pi_{\texttt{type}}\left(\Rsh^?(\dot{\tau}_x, \mathcal{S}tr, \$v_i)\right) = T_{x[y]}$ and $\pi_{\texttt{type}}\left(\Rsh^?(\dot{\tau}_u, \mathcal{S}tr, \$v_j)\right) = T_{u[v]}$.

Since an expression is typed with a set of security types and a set of writing effects (instead of a single type and a single writing effect), the constraints as well as the *lub's* and *glb's* operations of the old type system must be rewritten in order to account for this change. For instance, in the current running example, the hybrid type system types $\mathtt{u[v]}$ with $T_{u[v]}$ and $\mathtt{z}$ with $T_z = \{(\textsf{PRIM}^L, \texttt{tt})\}$. Therefore, in order to type $\mathtt{u[v] + z}$, the type system needs to combine two sets of security types paired up with runtime assertions. To this end, we make use of a function $\oplus_{\uplus}$, parameterized with a generic binary function

[2]Recall that the implicit look-up annotation is $\mathcal{S}tr$.

[3]Observe that e must either be a variable of the type system or a primitive value.

$\Cup$, that given two sets of elements paired up with runtime assertions, S_0 and S_1, generates a new set $S_0 \oplus_{\Cup} S_1$. If $(s, \omega) \in S_0 \oplus_{\Cup} S_1$, then, for every memory μ and reference r, $\mu, r \vDash \omega$ *iff* there are two pairs $(s_0, \omega_0) \in S_0$ and $(s_1, \omega_1) \in S_1$ such that $\mu, r \vDash (\omega_0 \wedge \omega_1)$ and $s = s_0 \Cup s_1$. Concretely, $T_{u[v]} \oplus_{\curlyvee} T_z = T_{u[v]}$. However, if we let $T'_z = \{(\mathsf{PRIM}^H, \mathtt{tt})\}$, $T_{u[v]} \oplus_{\curlyvee} T'_z = \{(\mathsf{PRIM}^H, \mathtt{tt})\}$.

In the rules that feature constraints, the hybrid type system tries to infer a dynamic assertion under which the corresponding expression is legal. For instance, when trying to type x[y] = u[v] + z, the hybrid type system tries to infer an assertion that is verified only if the level of the property that is being assigned is higher than or equal to the level of the right-hand side expression. Thus, we assume the existence of a function $When^?_{\Subset}$, parameterized with a generic order relation $\Subset$, that given two sets of elements paired up with runtime assertions, S_0 and S_1, generates an assertion $\omega = When^?_{\Subset}(S_0, S_1)$. The generated assertion describes the conditions under which there are two pairs $(s_0, \omega_0) \in S_0$ and $(s_1, \omega_1) \in S_1$ such that $s_0 \Subset s_1$ and $\omega_0 \wedge \omega_1$ holds. Formally, if $\omega = When^?_{\Subset}(S_0, S_1)$, then:

$$\forall_{\mu,r} \exists_{(s_0,\omega_0) \in S_0, (s_1,\omega_1) \in S_1} \; \mu, r \vDash \omega \Leftrightarrow \mu, r \vDash (\omega_0 \wedge \omega_1) \;\wedge\; s_0 \Subset s_1$$

For instance, in the current example: $When^?_{\preceq}(T_{x[y]}, T_{u[v]}) = (\$v_i \in \{p_0\}) || (\$v_j \in \{q_1\})$. If $\$v_i \in \{p_0\}$ then the property being assigned is *high* and the assignment is legal. If $\$v_j \in \{q_1\}$, then the value that is being assigned is *low* and, again, the assignment is legal.

The hybrid type system rewrites the program to be typed in order to dynamically check the assertions under which it is conditionally accepted. To this end, every conditionally typed expression is wrapped in a conditional expression that checks whether the assertion under which it was accepted holds. In order to simplify the specification, we make use of a syntactic function *wrap* that given an assertion ω, different from tt, and an expression e generates the expression $\omega\ ?\ (e) : (\_diverge())$, where $\_diverge()$ is a runtime function that always diverges. For instance, the program used as the running example is rewritten as follows: `_x_i = y, _x_j = v, (_x_i == "p_0" || _x_j == "q_1") ? (x[y] = u[v]) : (_diverge())`. If the type system is able to determine that a given constraint is always verified, it generates the assertion tt. In that case, *wrap* simply outputs the given expression.

In Figure 5.3, we present the hybrid type system for the imperative fragment of Core JavaScript. Typing judgements have the form: $\Gamma \vdash e \rightsquigarrow e'/e'' : T, L$, where (1) Γ is the typing environment, (2) e the expression to be typed, (3) e' a new expression semantically equivalent to e except for the executions that are considered illegal, (4) e'' an expression that bookkeeps the value to which e' evaluates, (5) T a set of security types paired up with runtime assertions, and (6) L a set of security levels paired up with runtime assertions. In the specification of the hybrid type system, we make use of a new function σ that given a set T of security types paired up with runtime assertions produces the set $\{(\sigma, \omega) \mid (\tau^\sigma, \omega) \in T\}$. Furthermore, given a set L of security levels paired up with runtime assertions, we use T^L for the set $\{(\dot{\tau}', \omega) \mid (\dot{\tau}, \omega_t) \in T \;\wedge\; (\sigma, \omega_l) \in L \;\wedge\; \omega = \omega_t \wedge \omega_l \;\wedge\; \dot{\tau}' = \dot{\tau}^\sigma\}$. Finally, we use T^ω for the set $\{(\dot{\tau}, \omega \wedge \omega') \mid (\dot{\tau}, \omega') \in T\}$ and V_F for the set of *falsy* values: $\{false, 0, undefined, null\}$.

In order to illustrate the difference in functioning between the static and the hybrid type systems, let us consider the Rule [Property Assignment]. In the typing of a property assignment, all of the three subexpressions e_0, e_1, and e_2 are typed with three sets of possible types T_0, T_1, and T_2 and three sets of possible writing effects L_0, L_1, and L_2. The runtime assertion ω_1 guarantees that the existence level of the actual property being assigned is higher than or equal to the levels of the resources on which the computation of e_0 and e_1 depends (thereby avoiding implicit flows), while ω_0 guarantees that its security level is higher than or equal to the level of the value that is assigned to it (thereby avoiding explicit flows). Finally, the constraint $\omega_2 \vee \omega'_2$ ensures that if the type of the receiver object does not include the $*$, then the property that is being assigned is in its domain. The instrumentation wraps the property assignment in a conditional expression that checks whether all of the three conditions hold.

The soundness of the hybrid TS is established by Theorems 5.2 and 5.3. The former states that the semantics of the instrumented program is contained in the semantics of the original one, while the latter states that the instrumented program is noninterferent. In the following, we use $\mu \simeq \mu'$ whenever μ and μ' coincide in all variables/properties available for the programmer and μ does not define mappings for variables/properties in $\mathcal{I}_C$.

Theorem 5.2 (Transparency)**.** *For any expression e, typing environment Γ, memory μ well-labeled by Σ, and reference r such that $\Gamma \vdash e \rightsquigarrow e'/e'' : T, L$ and $r \vdash \langle \mu, \Sigma, e'' \rangle \Downarrow \langle \mu'_f, \Sigma_f, v \rangle$; there exists a memory μ_f such that $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v \rangle$ and $\mu_f \simeq \mu'_f$.*

Theorem 5.3 (Noninterference)**.** *For any expression e and typing environment Γ, if $\Gamma \vdash e \rightsquigarrow e', e'' : T, L$, then e'' is noninterferent w.r.t. Γ.*

Val

$$\dfrac{\begin{array}{c} T = \{(\mathsf{PRIM}^{\perp}, \mathtt{tt})\} \\ L = \{(\top, \mathtt{tt})\} \end{array}}{\Gamma \vdash v \rightsquigarrow v/v : T, L}$$

This

$$\dfrac{\begin{array}{c} T = \{(\Gamma(\mathsf{this}), \mathtt{tt})\} \\ L = \{(\top, \mathtt{tt})\} \qquad e = \$v_i = \mathsf{this} \end{array}}{\Gamma \vdash \mathsf{this}^i \rightsquigarrow e/\$v_i : T, L}$$

Var

$$\dfrac{\begin{array}{c} T = \{(\Gamma(x), \mathtt{tt})\} \\ L = \{(\top, \mathtt{tt})\} \end{array}}{\Gamma \vdash x^i \rightsquigarrow \$v_i = x/\$v_i : T, L}$$

Binary Operation

$$\dfrac{\begin{array}{c} \forall_{i=0,1} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i \\ e' = e'_0, e'_1, \$v_j = e''_0 \ \mathsf{op}\ e''_1 \end{array}}{\Gamma \vdash e_0 \ \mathsf{op}^j\ e_1 \rightsquigarrow e'/\$v_j : T_0 \oplus_{\curlyvee} T_1, L_0 \oplus_{\sqcap} L_1}$$

Variable Assignment

$$\dfrac{\begin{array}{c} \Gamma \vdash e \rightsquigarrow e'/e'' : T, L \qquad L' = \{(lev(\Gamma(x)), \mathtt{tt})\} \\ L'' = L \oplus_{\sqcap} L' \quad When^{?}_{\preceq}(T, \{(\Gamma(x), \mathtt{tt})\}) = \omega \end{array}}{\Gamma \vdash x = e \rightsquigarrow e', wrap(\omega, x = e'')/e'' : T, L''}$$

Object Literal

$$\dfrac{\begin{array}{c} T = \{(\tau, \mathtt{tt})\} \ L = \{(\top, \mathtt{tt})\} \\ e' = \$v_i = \{\}^{\tau} \end{array}}{\Gamma \vdash \{\}^{\tau, i} \rightsquigarrow e'/\$v_i : T, L}$$

In Expression

$$\dfrac{\begin{array}{c} \forall_{i=0,1} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i \qquad L_P = \pi_{\mathtt{lev}}\left(\Rsh^{?} (T_0, P, e''_0)\right) \\ T = \{(\mathsf{PRIM}^{\perp}, \mathtt{tt})\}^{L_P \oplus_{\sqcup} lev(T_0) \oplus_{\sqcup} lev(T_1)} \end{array}}{\Gamma \vdash e_0 \ \mathsf{in}^P_j\ e_1 \rightsquigarrow e'_0, e'_1, \$v_j = e''_0 \ \mathsf{in}\ e''_1/\$v_j : T, L_0 \oplus_{\sqcap} L_1}$$

Property Deletion

$$\dfrac{\begin{array}{c} \Gamma \vdash e_0 \rightsquigarrow e'_0/e''_0 : T_0, L_0 \qquad L = \pi_{\mathtt{lev}}\left(\Rsh^{?} (T_0, \{p\}, e''_0)\right) \\ When^{?}_{\sqsubseteq}(lev(T_0), L) = \omega \qquad e' = e'_0, wrap(\omega, \$v_i = \mathsf{delete}\ e''_0.p) \end{array}}{\Gamma \vdash \mathsf{delete}^i\ e_0.p \rightsquigarrow e'/\$v_i : \{(\mathsf{PRIM}^{\perp}, \mathtt{tt})\}, L_0 \oplus_{\sqcap} L}$$

Sequence

$$\dfrac{\forall_{i=0,1} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i}{\Gamma \vdash e_0, e_1 \rightsquigarrow e'_0, e'_1/e''_1 : T_1, L_0 \oplus_{\sqcap} L_1}$$

Property Look-up

$$\dfrac{\begin{array}{c} \forall_{i=0,1} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i \quad L = L_0 \oplus_{\sqcap} L_1 \\ T = \pi_{\mathtt{type}}\left(\Rsh^{?} (T_0, P, e''_1)\right) \quad e = e'_0, e'_1, \$v_j = e''_0[e''_1] \end{array}}{\Gamma \vdash e_0[e_1, P]^j \rightsquigarrow e'/\$v_j : T^{lev(T_0) \oplus_{\sqcup} lev(T_1)}, L}$$

Property Assignment

$$\dfrac{\begin{array}{c} \forall_{i=0,1,2} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i \qquad LT = \Rsh^{?} (T_0, P, e''_1) \qquad L = \pi_{\mathtt{lev}}(LT) \qquad T = \pi_{\mathtt{type}}(LT) \\ When^{?}_{\preceq}(T_2, T) = \omega_0 \qquad When^{?}_{\sqsubseteq}(lev(T_0) \oplus_{\sqcup} lev(T_1), L) = \omega_1 \\ \omega_2 = \vee\{\omega \wedge (e''_1 \in dom(\dot{\tau})) \mid (\dot{\tau}, \omega) \in T_0 \ \wedge \ * \notin dom(\dot{\tau})\} \qquad \omega'_2 = \vee\{\omega \mid (\dot{\tau}, \omega) \in T_0 \ \wedge \ * \in dom(\dot{\tau})\} \end{array}}{\Gamma \vdash e_0[e_1, P] = e_2 \rightsquigarrow e'_0, e'_1, e'_2, wrap(\omega_0 \wedge \omega_1 \wedge (\omega_2 \vee \omega'_2), e''_0[e''_1] = e''_2)/e''_2 : T_2, L_0 \oplus_{\sqcap} L_1 \oplus_{\sqcap} L_2 \oplus_{\sqcap} L}$$

Conditional Expression

$$\dfrac{\begin{array}{c} \forall_{i=0,1,2} \cdot \Gamma \vdash e_i \rightsquigarrow e'_i/e''_i : T_i, L_i \qquad When^{?}_{\sqsubseteq}(lev(T_0), L_1 \oplus_{\sqcap} L_2) = \omega \\ e' = e'_0, wrap(\omega, e''_0 \ ? \ (e'_1, \$v_j = e''_1) : (e'_2, \$v_j = e''_2)) \qquad \omega_{\mathtt{tt}} = \neg(e''_0 \in V_F) \qquad \omega_{\mathtt{ff}} = (e''_0 \in V_F) \end{array}}{\Gamma \vdash e_0 \ ?^j \ (e_1) : (e_2) \rightsquigarrow e'/\$v_j : T_1^{\omega_{\mathtt{tt}}} \cup T_2^{\omega_{\mathtt{ff}}}, L_0 \oplus_{\sqcap} (L_1^{\omega_{\mathtt{tt}}} \cup L_2^{\omega_{\mathtt{ff}}})}$$

Figure 5.3: Hybrid Typing Secure Information Flow in Core JavaScript

Theorem 5.4 characterizes the precision of the hybrid TS. It shows that, given two expressions $\hat{e}$ and e that only differ in look-up annotations, whenever $\hat{e}$ is typable using the purely static TS and it converges, then e is typable using the hybrid TS and its instrumentation also converges. Hence, the theorem shows that the changing of look-up annotations of an expression $\hat{e}$, typable using the purely static TS, always yields an expression e, typable using the hybrid TS, whose evaluation never diverges due to the failure of runtime assertions.

Theorem 5.4 (Precision). *For any two expressions $\hat{e}$ and e, typing environment Γ, and memory μ well-labeled by Σ such that: $\hat{e} \equiv e$, $\Gamma = \Rsh (\Sigma(\#glob))$, $\Gamma \vdash \hat{e} : \tau, \sigma$, and $\#glob \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v \rangle$; there exists a memory μ'_f such that: $\Gamma \vdash e \rightsquigarrow e'/e'' : T, L$ and $\#glob \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu'_f, \Sigma_f, v \rangle$.*

5.5 Related Work

5.5.1 Static Type Systems for Securing Information Flow

Since the seminal work of Volpano *et al* [Volpano 1996] on typing secure information flow in a simple imperative language, TSs for IFC have been proposed for a wide variety of languages, ranging from functional [Pottier 2003] to Java-like object-oriented languages [Banerjee 2002]. To the best of our knowledge, our TS is the first one that addresses the particular features of JavaScript in the context of IFC.

5.5.2 Monitoring Secure Information Flow

The increasing popularity of dynamic languages has motivated further research on runtime mechanisms for IFC, such as *information flow monitors.* In contrast to *purely dynamic monitors* [Austin 2009, Austin 2010, Austin 2012] that do not rely on any kind of static analysis, *hybrid monitors* [Guernic 2007, Shroff 2007, Venkatakrishnan 2006], use static analysis to reason about the implicit flows that arise due to untaken execution paths. Furthermore, some hybrid monitors also use static analyses to boost performance. For instance, Moore *et al* [Moore 2011] show how to combine monitoring and static analysis so as to reduce the number of variables whose levels are tracked at runtime. Interestingly, Russo *et al*[Russo 2010] prove that hybrid monitors are more permissive than both purely dynamic and purely static enforcement mechanisms. Their result supports the need for mechanisms which combine static and dynamic analysis like our own. However, unlike hybrid monitors, the hybrid TS we propose does **not** require any kind of runtime tracking of security levels, since the inlined conditions feature the actual values that are computed by the program rather than their levels.

5.5.3 Gradual Typing Secure Information Flow

Recently, *gradual security typing* [Disney 2011, Fennell 2013] has been proposed as a way to combine runtime monitoring and static analysis in order to cater for controlled forms of *polymorphism.* Concretely, the programmer is expected to introduce runtime casts in points where values of a pre-determined security type are expected. "The type system statically guarantees adherence to the [security] policies on the static side of a cast, whereas the runtime system checks the policies on the dynamic side"[Fennell 2013]. This approach could be used for the typing of arbitrary property look-ups. However, this would necessarily imply partial tracking of security levels, which our solution does not require.

5.5.4 Static Analysis for Securing JavaScript Applications

Due to the complexity of JavaScript semantics, most mechanisms for preventing security violations spawned by client-side JavaScript code have focused on isolation properties [FBJS , Maffeis 2009, Crockford , Politz 2011], which are easier to enforce than noninterference [Goguen 1982]. The analyses presented in [Maffeis 2009] and [Politz 2011] deal in different ways with the issue of property look-ups featuring arbitrary expressions. While the authors of [Maffeis 2009] consider a subset of the language that does not include this kind of look-up expression, the TS presented in [Politz 2011] overapproximates the set of properties to which these arbitrary expressions may evaluate. We believe that the idea illustrated by the hybrid TS could be applied both to [Maffeis 2009] and [Politz 2011] in order to increase their permissiveness.

5.5.5 Static Analysis for JavaScript

Thiemann [Thiemann 2005] (from whom we borrow the idea of *default type*) has proposed a TS that guarantees *termination* and *progress* for a fragment of JavaScript, which does not account for objects whose domain may change at runtime. This is a severe restriction since it precludes a feature of the language commonly used in practice [Richards 2010]. To overcome this issue, Anderson *et al* [Anderson 2005] have proposed a type inference algorithm that allows objects "to evolve in a controlled manner" by classifying their properties as *definite* or *potential.* This additional information could be used by the static TS to distinguish *property creations* from *property updates*, thereby relaxing the constraints imposed on property updates, which would not need to take into account the existence level of the updated property.

Chapter 6

An Extensible Monitored Semantics for Securing Web APIs

Contents

Although JavaScript can be used as a general-purpose programming language, many JavaScript programs are designed to be executed in a browser in the context of a web page. Such programs often interact with the web page in which they are included, as well as the browser itself, through Application Programming Interfaces (APIs). Some APIs are fully implemented in JavaScript, whereas others are built with a mix of different technologies, which can be exploited to conceal sophisticated security violations. Thus, understanding the behavior of client-side web applications as well as proving their compliance with a given security policy requires cross-language reasoning that is often far from trivial.

The size, complexity, and number of commonly used APIs poses an important challenge to any attempt at formally reasoning about the security of JavaScript programs [Guha 2012]. To tackle this problem, we propose a methodology for extending JavaScript monitored semantics. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, we make it possible to prove that a security monitor is still noninterferent when extending it with a new API, without having to revisit the whole model.

Generally, an API can be viewed as a particular set of specifications that a program can follow to make use of the resources provided by another particular application. For client-side JavaScript programs, this definition of API applies both to:

- interfaces of services that are provided to the program by the environment in which it executes, namely the web browser (for instance, the DOM, the XMLHttpRequest, and the W3C Geolocation APIs);
- interfaces of JavaScript libraries that are explicitly included by the programmer (for instance, jQuery, Prototype.js, and Google Maps Image API).

In the context of this work, the main difference between these two types of APIs is that in the former case their semantics escapes the JavaScript semantics, whereas in the latter it does not. The methodology proposed here was designed as a generic way of extending security monitors to deal with the first type of APIs. Nevertheless, we can also apply it to the second type whenever we want to execute the library's code in the original JavaScript semantics instead of the monitored semantics.

6.1 An Extensible Semantics for Core JavaScript

At the formal level, in order to model the execution of APIs whose semantics may eventually escape the JavaScript semantics, we extend the semantics of Core JavaScript ($\Downarrow$) with alternative rules for property look-ups and method calls. These alternative rules allow for the execution of arbitrary external APIs. Concretely, upon the invocation of a method, the new semantics checks whether it is a standard method or rather a method belonging to an API. In the former case, the semantics proceeds as before, whereas in the latter it uses the semantics of that particular API to compute its return value. Likewise, when looking-up the value of an object's property, the semantics checks whether that property look-up should be handled by an external API (rather than the JavaScript engine) in which case it uses the semantics of that particular API to compute the value yielded by that property look-up.

Formally, we model an API as a triple $\langle \mathcal{D}, \mathcal{A}, \mathcal{R} \rangle$ consisting of: **(1)** a semantic domain $\mathcal{D}$ that captures the current state of the API, **(2)** a set $\mathcal{A}$ of functions that operate on that domain, that we call API methods, and **(3)** a mapping $\mathcal{R}$, that we call API register, used to determine when to apply each API method. An API method can be seen as a function that, given a sequence of values, updates the current state of the API and produces a new value. Therefore, we model an API method api as a relation of the form: $\langle \nu, \overrightarrow{v} \rangle^i \ \mathsf{api}\ \langle \nu', v \rangle^\alpha$ where: **(1)** $\nu \in \mathcal{D}$ is the current state of the API, **(2)** $\overrightarrow{v}$ the sequence of values given as input optionally annotated with an index i, **(3)** $\nu' \in \mathcal{D}$ the state of the API after the execution of api, **(4)** v the produced value, and **(5)** α an internal event used by the security monitor and explained in Section 6.2.

The extended semantics intercepts property look-ups and methods calls. The first two subexpressions (in evaluation order) of these two kinds of expressions evaluate to a reference and a string, respectively. These two values together with the kind of the current expression are used by the API register to determine whether its evaluation should trigger the execution of an API and, if so, which API method to apply. Hence, when executing a method named m of an object o pointed to by a reference r_o, the semantics first checks whether the tuple $\langle r_o, m, \text{“MC”} \rangle$ is in the domain of the API register $\mathcal{R}$, in which case the corresponding API method is applied. Likewise, when looking up the value of o's property m, the semantics checks whether $\langle r_o, m, \text{“LU”} \rangle \in \mathit{dom}(\mathcal{R})$, in which case it uses the corresponding API. The strings “MC” and “LU” are used by the API register to differentiate property look-ups from method calls.

Figure 6.1 presents the semantics of Core JavaScript extended with an arbitrary API $\langle \mathcal{D}, \mathcal{A}, \mathcal{R} \rangle$. To this end, both initial and final configurations must be extended with an additional API state. Since the API register is assumed not to change during execution, it is left implicit. That is, we do not explicitly include it in the semantic relation. Therefore, transitions of the extended Core JavaScript semantics have the following form: $r \vdash \langle \mu, e \mid \nu \rangle \Downarrow \langle \mu', v \mid \nu' \rangle$, where: ν and ν' are the initial and final API states. The remaining elements keep their original meanings.

6.1.1 An API for Using Priority Queues

Consider, for instance, an API for creating and manipulating priority queues accessible through the reference $\#r_Q$ initially stored in the global variable $\mathit{queueAPI}$. This API features the methods: **(1)** $\mathit{queueAPI.queue}()$ for creating a new priority queue, **(2)** $q.\mathit{push}(\mathit{el}, \mathit{pri})$ for adding the element el with priority pri to the queue q, **(3)** $q.\mathit{pop}()$ for removing the element with the highest priority from the priority queue q, and **(4)** $q.\mathit{empty}()$ for checking whether or not q is empty. To extend Core JavaScript semantics with this API, we have to define its formal specification. That is, we must define the corresponding semantic domain, API methods, and API register.

At the formal level, we model: **(1)** a state of the Queue API as a mapping from references to priority queues, **(2)** a priority queue Q as a list of nodes, $n_0 :: n_1 :: ... :: n_n$, and **(3)** a priority queue node n as a pair $\langle v, i \rangle$ consisting of a value and a number indicating its priority. The nodes in the list of a priority queue are ordered by priority. In the following, we use the notation $\mathsf{priority}_\downarrow(Q)$ for the priority of the node with lowest priority in Q and $\mathsf{priority}_\uparrow(Q)$ for the priority of the node with highest priority in Q. Figure 6.2 gives the formal specification of the methods that compose this API.

The formal semantics of the Queue API assumes that the references used by the Queue API do not overlap with the references used by the standard semantics. Specifically, the co-domain of the allocator used in the semantics of the Queue API, fresh_Q, is assumed not to overlap with that of the allocator of standard Core JavaScript. We denote by $\mathcal{R}ef_Q$ the set of references for the use of the Queue API. The

Property Look-up

$$\frac{\begin{array}{c} r \vdash \langle \mu, e_0 \mid \nu\rangle \Downarrow \langle \mu_0, r_0 \mid \nu_0\rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0\rangle \Downarrow \langle \mu_1, m_1 \mid \nu_1\rangle \\ \langle r_0, m_1, \text{“PLU”}\rangle \notin dom(\mathcal{R}) \qquad \langle \mu_1, r_0, m_1\rangle\ \mathcal{R}_{Proto}\ r' \\ r' \neq null \Rightarrow v = \mu_1(r')(m_1) \qquad r' = null \Rightarrow v = undefined \end{array}}{r \vdash \langle \mu, e_0[e_1]^i \mid \nu\rangle \Downarrow \langle \mu_1, v \mid \nu\rangle}$$

External Property Look-up

$$\frac{\begin{array}{c} r \vdash \langle \mu, e_0 \mid \nu, \rangle \Downarrow \langle \mu_0, r_0 \mid \nu_0\rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0\rangle \Downarrow \langle \mu_1, m_1 \mid \nu_1\rangle \\ \langle r_0, m_1, \text{“PLU”}\rangle \in dom(\mathcal{R}) \qquad \mathsf{api} = \mathcal{R}(r_0, m_1, \text{“PLU”}) \qquad \langle \nu_1, r_0 :: m_1\rangle^i\ \mathsf{api}\ \langle \nu', v\rangle \end{array}}{r \vdash \langle \mu, e_0[e_1]^i \mid \nu\rangle \Downarrow \langle \mu_1, v \mid \nu'\rangle}$$

Method Call

$$\frac{\begin{array}{c} r \vdash \langle \mu, e_0 \mid \nu\rangle \Downarrow \langle \mu_0, r_0 \mid \nu_0\rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0\rangle \Downarrow \langle \mu_1, m_1 \mid \nu_1\rangle \\ r \vdash \langle \mu_1, e_2 \mid \nu_1\rangle \Downarrow \langle \mu_2, v_2 \mid \nu_2\rangle \qquad \langle r_0, m_1, \text{“MC”}\rangle \notin dom(\mathcal{R}) \qquad \langle \mu_2, r_0, m_1\rangle\ \mathcal{R}_{Proto}\ r_m \\ r_f = \mu_2(r_m)(m_1) \qquad \langle \mu_2, r_f, v_2, r_0\rangle\ \mathcal{R}_{NewScope}\ \langle \hat{\mu}, \hat{e}, \hat{r}\rangle \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e} \mid \nu_2\rangle \Downarrow \langle \mu', v \mid \nu'\rangle \end{array}}{r \vdash \langle \mu, e_0[e_1](e_2)^i \mid \nu\rangle \Downarrow \langle \mu', v \mid \nu'\rangle}$$

External Method Call

$$\frac{\begin{array}{c} r \vdash \langle \mu, e_0 \mid \nu\rangle \Downarrow \langle \mu_0, r_0 \mid \nu_0\rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0\rangle \Downarrow \langle \mu_1, m_1 \mid \nu_1\rangle \\ r \vdash \langle \mu_1, e_2 \mid \nu_1\rangle \Downarrow \langle \mu_2, v_2 \mid \nu_2\rangle \qquad \langle r_0, m_1, \text{“MC”}\rangle \in dom(\mathcal{R}) \\ \mathsf{api} = \mathcal{R}(r_0, m_1, \text{“MC”}) \qquad \langle \nu_2, r_0 :: m_1 :: v_2\rangle^i\ \mathsf{api}\ \langle \nu', v\rangle \end{array}}{r \vdash \langle \mu, e_0[e_1](e_2) \mid \nu\rangle \Downarrow \langle \mu', v \mid \nu'\rangle}$$

Figure 6.1: Extending the Semantics of Core JavaScript

Queue

$$\frac{r = fresh_Q(\nu, i) \qquad \nu' = \nu\,[r \mapsto \varepsilon]}{\langle \nu, \#r_Q :: \text{“queue”}\rangle^i\ \mathsf{queue}\ \langle \nu', r\rangle}$$

Pop

$$\frac{\nu(r) = \langle v, i\rangle :: Q \qquad \nu' = \nu\,[r \mapsto Q]}{\langle \nu, r :: \text{“pop”}\rangle\ \mathsf{pop}\ \langle \nu', v\rangle}$$

Push

$$\frac{\begin{array}{c} \nu(r) = Q_L :: Q_R \qquad \mathsf{priority}_{\downarrow}(Q_L) \geq j \\ j > \mathsf{priority}_{\uparrow}(Q_R)\ \vee\ Q_R = \varepsilon \\ \nu' = \nu\,[r \mapsto Q_L :: \langle v, j\rangle :: Q_R] \end{array}}{\langle \nu, r :: \text{“push”} :: v :: j\rangle\ \mathsf{push}\ \langle \nu', v\rangle}$$

Empty

$$\frac{\begin{array}{c} \nu(r) = \varepsilon \Rightarrow v = \mathtt{tt} \\ \nu(r) \neq \varepsilon \Rightarrow v = \mathtt{ff} \end{array}}{\langle \nu, r :: \text{“empty”}\rangle\ \mathsf{empty}\ \langle \nu, v\rangle}$$

Figure 6.2: A Priority Queue API

$$\text{EXTERNAL PROPERTY LOOK-UP}\quad \frac{\begin{array}{c}\forall_{i=0,1}\ r,\sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \mid \nu_i, \Xi_i\rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \mid \nu_{i+1}, \Xi_{i+1}\rangle \\ \langle r_0, m_1, \text{“PLU”}\rangle \in dom(\mathcal{R}) \qquad (\mathsf{api}, \mathsf{api}_{lab}) = \mathcal{R}(v_0, v_1, \text{“PLU”}) \\ \langle \nu_2, v_0 :: v_1\rangle^i\ \mathsf{api}\ \langle \nu', v\rangle^\alpha \qquad \langle \Xi_2, \sigma_0 :: \sigma_1\rangle^\alpha\ \mathsf{api}_{lab}\ \langle \Xi', \sigma\rangle\end{array}}{r,\sigma_{pc} \vdash \langle \mu_0, e_0[e_1]^i, \Sigma_0 \mid \nu_0, \Xi_0\rangle \Downarrow_{IF} \langle \mu_2, v, \Sigma_2, \sigma \mid \nu', \Xi'\rangle}$$

$$\text{EXTERNAL METHOD CALL}\quad \frac{\begin{array}{c}\forall_{i=0,1,2}\ r,\sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \mid \nu_i, \Xi_i\rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \mid \nu_{i+1}, \Xi_{i+1}\rangle \\ \langle r_0, m_1, \text{“PLU”}\rangle \in dom(\mathcal{R}) \qquad (\mathsf{api}, \mathsf{api}_{lab}) = \mathcal{R}(v_0, v_1, \text{“PLU”}) \\ \langle \nu_3, v_0 :: v_1 :: v_2\rangle^i\ \mathsf{api}\ \langle \nu', v\rangle^\alpha \qquad \langle \Xi_3, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^\alpha\ \mathsf{api}_{lab}\ \langle \Xi', \sigma\rangle\end{array}}{r,\sigma_{pc} \vdash \langle \mu_0, e_0[e_1](e_2)^i, \Sigma_0 \mid \nu_0, \Xi_0\rangle \Downarrow_{IF} \langle \mu_3, v, \Sigma_3, \sigma \mid \nu', \Xi'\rangle}$$

Figure 6.3: Extending the Semantics of Core JavaScript

API register is given below:

$$\mathcal{R}(v_0, v_1, m) = \begin{cases} \mathsf{queue} & \text{if } v_1 = \text{“queue”} \wedge v_0 = \#r_Q \wedge m = \text{“MC”} \\ \mathsf{pop} & \text{if } v_1 = \text{“push”} \wedge v_0 \in \mathcal{R}ef_Q \wedge m = \text{“MC”} \\ \mathsf{push} & \text{if } v_1 = \text{“pop”} \wedge v_0 \in \mathcal{R}ef_Q \wedge m = \text{“MC”} \\ \mathsf{empty} & \text{if } v_1 = \text{“empty”} \wedge v_0 \in \mathcal{R}ef_Q \wedge m = \text{“MC”} \end{cases}$$

6.2 A Secure Extensible Monitor for Core JavaScript

Having shown how to extend Core JavaScript semantics in order to take into account the execution of APIs that may take place outside the JavaScript engine, we now show how to extend its monitored version presented in Chapter 4. We define the monitored semantics of external APIs in the style of Russo et al. [Russo 2010, Sabelfeld 2009]. Each API state ν is paired up with an abstract state Ξ, that we call API labelling, which defines, for each security level σ, the resources in ν visible for an attacker at level σ. Likewise, each API method is paired up with a monitor counterpart that defines how the API labelling Ξ should be updated after the execution of the API. The monitor method uses the internal event generated by the original method as well as the security levels of its arguments to determine how the API labelling should be updated. Hence, an API monitor method is modelled as a relation of the form $\langle \Xi, \overrightarrow{\sigma}\rangle^\alpha \mathsf{api}_{lab}\ \langle \Xi', \sigma\rangle$, where: **(1)** Ξ is the current API labelling, **(2)** $\overrightarrow{\sigma}$ the levels of the arguments given as input to the API method, **(3)** α the internal event generated by the API method, **(4)** Ξ' the API labelling after the execution of the API method, and **(5)** σ the security level associated with its produced value. The API register is modified in such a way that it outputs both the original method **and** the monitor method.

A configuration of the monitored semantics for extended Core JavaScript is obtained by adding both to the initial and final configurations of the original monitor an API state ν and an API labelling Ξ. The rules of the extended monitored semantics, $\Downarrow_{IF}$, presented in Figure 6.3, have the form $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi\rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \mid \nu', \Xi'\rangle$, where: **(1)** ν and ν' are the initial and final API states and **(2)** Ξ and Ξ' are the initial and final API labelings. The remaining elements keep their original meanings. We only present the rules that interact with the API state.

6.2.1 Secure APIs

For the extended monitor to be noninterferent one must impose some constraints on the API methods that can be invoked. Definitions 6.1 and 6.3 formalize these requirements. Definition 6.1 states that an API method is *confined* if it only creates/updates resources whose levels are higher than or equal to the level of the values that were used to decide which API method to apply. Observe that, since these two levels are higher than or equal to the level of the context in which the API is called, this property also guarantees that the execution of an API does neither create nor change resources whose levels are not higher than or equal to the level of the current context. In the following we assume the existence of a low-equality relation $\sim_{api}$ parameterizable in a security level σ that defines what resources of the API state are visible at level σ.

Definition 6.1 (Confined API Method)**.** *A pair* $(\mathsf{api}, \mathsf{api}_{lab})$ *consisting of an API method and its labeled counterpart is said to be confined if for any API state* ν *and labelling* Ξ*, any sequence of values* $\overrightarrow{v}$ *respectively labeled by a sequence of levels* $\overrightarrow{\sigma}$*, and any security level* σ*, such that:* **(1)** $\langle \nu, \overrightarrow{v} \rangle^i \; \mathsf{api} \; \langle \nu', v \rangle^\alpha$*,* **(2)** $\langle \Xi, \overrightarrow{\sigma} \rangle^\alpha \; \mathsf{api}_{lab} \; \langle \Xi', \sigma' \rangle$*, and* **(3)** $\sqcup \overrightarrow{\sigma} \not\sqsubseteq \sigma$*; then, it follows that:* $\nu, \Xi \; \sim^{\sigma}_{api} \; \nu', \Xi'$ *and* $\sigma' \not\sqsubseteq \sigma$*.*

Definition 6.3 states that an API relation is *noninterferent* if whenever it is executed on two low-equal API states, it produces two low-equal API states and either the two output values are both visible and coincide or they are both invisible. Informally, an API register $\mathcal{R}$ is said to be noninterferent, written $\mathbf{NI}(\mathcal{R})$ if all the API methods in its codomain are noninterferent. In the following, we use a low-equality for sequences of labeled values that states that two lists of labeled values are low-equal with respect to a given security level σ, if for each position of both sequences, either the two values in that position coincide, or their levels are both $\not\sqsubseteq \sigma$. Definition 6.2 formalizes this notion. Given a sequence $\overrightarrow{v}$, we use $\overrightarrow{v}(i)$ for the ith element of $\overrightarrow{v}$ and $|\overrightarrow{v}|$ for its number of elements.

Definition 6.2 (Low-Equality for Sequences)**.** *Two lists of values* $\overrightarrow{v}_0$ *and* $\overrightarrow{v}_1$ *respectively labeled by two lists of security levels* $\overrightarrow{\sigma}_0$ *and* $\overrightarrow{\sigma}_1$ *are said to be low-equal w.r.t. a security level* σ*, written* $\overrightarrow{v}_0, \overrightarrow{\sigma}_0 \sim_\sigma \overrightarrow{v}_1, \overrightarrow{\sigma}_1$ *if the following hold:* **(1)** $\forall_{0 \leq i < n} \; \overrightarrow{\sigma}_0(i) \sqcap \overrightarrow{\sigma}_1(i) \sqsubseteq \sigma \Rightarrow \overrightarrow{v}_0(i) = \overrightarrow{v}_1(i) \; \wedge \; \overrightarrow{\sigma}_0(i) = \overrightarrow{\sigma}_1(i) \sqsubseteq \sigma$*,* **(2)** $\forall_{n < i < |\overrightarrow{v}_0|} \; \overrightarrow{\sigma}_0(i) \not\sqsubseteq \sigma$*, and* **(3)** $\forall_{n < j < |\overrightarrow{v}_1|} \; \overrightarrow{\sigma}_1(j) \not\sqsubseteq \sigma$ *where* $n = \mathsf{min}(|\overrightarrow{v}_0|, |\overrightarrow{v}_1|)$*.*

Definition 6.3 (Noninterferent API Method)**.** *A pair* $(\mathsf{api}, \mathsf{api}_{lab})$ *consisting of an API method and its labeled counterpart is said to be noninterferent, written* $\mathbf{NI}(\mathsf{api}, \mathsf{api}_{lab})$*, if it is confined and for any two API states* ν_0 *and* ν_1 *and labelings* Ξ_0 *and* Ξ_1*, any two sequences of values* $\overrightarrow{v}_0$ *and* $\overrightarrow{v}_1$ *labeled by* $\overrightarrow{\sigma}_0$ *and* $\overrightarrow{\sigma}_1$*, and any security level* σ *such that:* **(1)** $\overrightarrow{v}_0, \overrightarrow{\sigma}_0 \; \sim_\sigma \; \overrightarrow{v}_1, \overrightarrow{\sigma}_1$*,* **(2)** $\nu_0, \Xi_0 \sim_\sigma \nu_1, \Xi_1$*,* **(3)** $\langle \nu_0, \overrightarrow{v}_0 \rangle^i \; \mathsf{api} \langle \nu_0', v_0 \rangle^\alpha$*,* **(4)** $\langle \Xi_0, \overrightarrow{\sigma}_0 \rangle^\alpha \; \mathsf{api}_{lab} \; \langle \Xi_0', \sigma_0 \rangle$*,* **(5)** $\langle \nu_1, \overrightarrow{v}_1 \rangle^i \; \mathsf{api} \; \langle \nu_1', v_1 \rangle^\alpha$*,* **(6)** $\langle \Xi_1, \overrightarrow{\sigma}_1 \rangle^\alpha \; \mathsf{api}_{lab} \; \langle \Xi_1', \sigma_1 \rangle$*; then:* $\nu_0', \Xi_0' \sim_\sigma \nu_1', \Xi_1'$ *and* $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$*.*

6.2.2 A Secure Queue API

We now define the monitor methods corresponding to each method of the Queue API and prove the corresponding monitored Queue register to be noninterferent. Before proceeding to the formal specification of the monitor methods, we give an informal description of the security leaks entailed by the specific semantics of the Queue API.

6.2.2.1 Challenges in Information Flow Control for Priority Queues

The range of operations offered by the Queue API can be exploited to encode security leaks via the order by which new nodes are inserted in a queue. For instance, the program presented below creates a new queue, pushes the string "a" with priority 1 into this queue, and, depending on the value a secret variable h, additionally pushes a string "b" with priority 2. Hence, whenever $h \notin V_F$, "b" is placed on top of the queue. Then, the program pops the top of the queue and compares it with "a", in which case it sets the low variable l (originally set to 1) to 0.

```
q = queueAPI.queue(),
q.push("a", 1),
l = 1,
h ? q.push("b", 2),
q.pop() == "a" : l = 0
```

Observe that, depending on the original value of the *high* variable h, the *low* variable l may be either set to 0 or 1 after the execution of this program. However, if we swap the priorities of the two nodes as in the following program the security leak illustrated by the previous example disappears.

```
q = queueAPI.queue(),
q.push("a", 2),
l = 1,
h ? q.push("b", 1),
q.pop() == "a" : l = 1
```

In contrast to the second example, the first example exploits the fact that inserting a node in a queue changes the positions of pre-existing nodes with lower priority. If these nodes are visible (which is the case in the first example), the changes in their positions are also visible. Analogously, inserting a node with a secret priority in a queue containing visible nodes with lower priorities will cause the positions of these nodes to change. These changes will also be visible, which we illustrate in the program below.

```
q = queueAPI.queue(),
q.push("a", 1),
q.push("b", h),
l = 1,
q.pop() == "a" : l = 0
```

After the execution of this program, depending on the initial value of the *high* variable h, the *low* variable l is either assigned to 0 or to 1, thus revealing information about the original value of h. More specifically, if after running the program l is set to 1, then we can conclude that the initial value of h is > 1. Likewise, if l is set to 1, then the initial value of h is ≤ 1.

In order to cope with information flows illustrated in the examples above, the monitored Queue API blocks the insertion of a node in a secret context or a node with a secret priority, whenever there is a visible pre-existing node whose priority is lower than that of the node to be inserted. Observe that this rule renders the execution of the first program illegal whenever $h \notin V_F$ and the execution of the third program illegal whenever $h > 1$.

Another way to encode information flows using the Queue API consists in inserting a node in a *high* context and then checking (in a *low*) context whether that queue is empty as in the program below:

```
q = queueAPI.queue(),
h ? q.push("a", 1),
l = q.empty()
```

Observe that after the execution of this program l is set to `tt` whenever $h \in V_F$ and l is set to `ff` whenever $h \notin V_F$. Therefore, the final value of l depends on the initial value of h. To prevent this type of information flow, the monitored Queue API associates a structure security level with each queue. New elements can only be pushed into a queue in contexts whose levels are lower than or equal to its corresponding structure security level. Accordingly, the level associated with the invocation of the API method *empty* in a queue q is q's structure security level. For instance, for the monitor not to block the execution of the program above in a memory such that $h \notin V_F$, q must have a *high* structure security level. Hence, the monitored execution of the final assignment will always upgrade the security level of l to *high*.

6.2.2.2 An Attacker Model for Priority Queues

In order to formally characterise what part of the API state an attacker at a given security level can observe, we must define a low-equality relation $\sim_Q$ for Queue API states parameterizable in an arbitrary security level σ. Two Queue API states ν and ν' respectively labeled by Ξ and Ξ' are related by $\sim_Q^\sigma$ if an attacker at level σ cannot distinguish the two of them. To this end, we start by defining a security labelling for queues as a partial function $\Xi : \mathcal{R}ef_Q \to \mathcal{L} \times \mathcal{L} \times 2^{\mathcal{L}}$ that associates a queue reference with a 3-tuple consisting of:

1. The level of the context in which the queue was created – denoted *queue level*;
2. The structure security level of the queue;
3. The list containing the security levels of the queue's nodes – denoted *queue levels list*.

Given a queue reference r and labelling Ξ, $\Xi(r) = \langle \sigma_q, \sigma_s, \overrightarrow{\sigma} \rangle$, where: **(1)** σ_q is the queue level, **(2)** σ_s is the structure security level, and **(3)** $\overrightarrow{\sigma}$ is the queue levels list. For clarity, given a queue q pointed to by a reference r and a labelling Ξ, we denote by $\Xi(r)$.queue, $\Xi(r)$.struct, and $\Xi(r)$.qnodes its queue level, its structure security level, and its queue levels list, respectively.

The low-projection of a Queue API state ν labelled by Ξ at security level σ corresponds to the part of ν that is visible for an attacker at level σ. Informally, given a Queue API state ν labelled by Ξ, an attacker at level σ can see: **(1)** the queue references whose queue levels are $\leq \sigma$, **(2)** the number of nodes of visible queues whose structure security level is $\leq \sigma$, and **(3)** the nodes of visible queues associated whose corresponding level in the respective queue level list is $\leq \sigma$. Observing a node of queue means seeing the value that it stores as well as the position it occupies in the corresponding queue.

Queue

$$\frac{\sigma' = \sigma_0 \sqcup \sigma_1 \qquad \Xi' = \Xi\left[r \mapsto \langle \sigma', \sigma_s, \varepsilon\rangle\right]}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{r,\sigma_s}\ \mathsf{queue}_{lab}\ \langle \Xi', \sigma'\rangle}$$

Pop

$$\frac{\begin{array}{c}\Xi(r).\mathsf{qnodes} = \sigma :: \vec{\sigma} \qquad \sigma' = \sigma_0 \sqcup \sigma_1 \\ \sigma' \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r).\mathsf{qnodes}(0) \\ \Xi' = \Xi\left[r \mapsto \langle \Xi(r).\mathsf{queue}, \Xi(r).\mathsf{struct}, \vec{\sigma}\rangle\right]\end{array}}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{r}\ \mathsf{pop}_{lab}\ \langle \Xi, \sigma' \sqcup \sigma\rangle}$$

Push

$$\frac{\begin{array}{c}\Xi(r).\mathsf{qnodes} = \vec{\sigma}_0 :: \vec{\sigma}_1 \qquad |\vec{\sigma}_0| = i \\ \sigma' = \sigma_0 \sqcup \sigma_1 \qquad \sigma = \sigma' \sqcup \sigma_2 \sqcup \sigma_4 \\ \sigma' \sqsubseteq \Xi(r).\mathsf{struct} \qquad |\vec{\sigma}_1| \neq 0 \Rightarrow \sigma \sqsubseteq \vec{\sigma}_1(0) \\ \Xi' = \Xi\left[r \mapsto \langle \Xi(r).\mathsf{queue}, \Xi(r).\mathsf{struct}, \vec{\sigma}_0 :: \sigma :: \vec{\sigma}_1\rangle\right]\end{array}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 :: \sigma_3\rangle^{r,i}\ \mathsf{push}_{lab}\ \langle \Xi', \sigma' \sqcup \sigma_2\rangle}$$

Empty

$$\frac{\sigma = \Xi(r).\mathsf{struct} \sqcup \sigma_0 \sqcup \sigma_1}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{r}\ \mathsf{empty}_{lab}\ \langle \Xi, \sigma\rangle}$$

Figure 6.4: Monitor Methods for the Queue API

Definition 6.4 (Low-Projection and Low-Equality for Queues). *The low-projection of a Queue API state ν w.r.t. a security level σ and a labeling Ξ is given by:*

$$\begin{array}{rl}\nu \upharpoonright^{\Xi,\sigma} = & \{(r, \Xi(r).\mathsf{queue}) \mid \Xi(r).\mathsf{queue} \sqsubseteq \sigma\} \\ & \cup\ \{(r, n, \Xi(r).\mathsf{struct}) \mid\ \Xi(r).\mathsf{queue} \sqcup \Xi(r).\mathsf{struct} \sqsubseteq \sigma \wedge |\nu(r)|\} \\ & \cup\ \{(r, v, i, \Xi(r).\mathsf{qnodes}(i)) \mid\ \Xi(r).\mathsf{queue} \sqcup \Xi(r).\mathsf{qnodes}(i) \sqsubseteq \sigma \wedge \nu(r)(i) = (v, j)\}\end{array}$$

Two Queue API states ν_0 and ν_1, respectively labeled by Ξ_0 and Ξ_1 are said to be low-equal at security level σ, written $\nu_0, \Xi_0 \sim_Q^{\sigma} \nu_1, \Xi_1$ if they coincide in their respective low-projections, $\nu_0 \upharpoonright^{\Xi_0,\sigma} = \nu_1 \upharpoonright^{\Xi_1,\sigma}$.

6.2.2.3 Enforcing Secure Information Flow in the Queue API

Figure 6.4 presents the monitor methods for the Queue API. As discussed above, the basic restrictions that the monitored Queue API methods enforce are the following:

- One cannot change the structure of a queue (either by pushing new elements into it or popping elements out of it) with a visible structure security level inside an invisible context;
- One cannot push an invisible element into a queue containing visible elements with lower priorities.

The first constraint is enforced by both rules [Pop] and [Push], whereas the second is only enforced by [Push]. All the four monitor methods assume that the corresponding input annotation includes the reference of the queue on which the corresponding API method was invoked. Additionally, queue_{lab} assumes its input to be annotated with the structure security level of the corresponding queue, whereas push_{lab} assumes its input to be annotated with the position in which the new node is going to be inserted (for instance, if the annotation is 3, the new node will occupy the third position, meaning that it will have 3 nodes on its left).

6.3 IFlow Signatures for Securing Web APIs

A call to an external API cannot be instrumented in the same way one instruments a normal JavaScript method, simply because the code of APIs methods is usually not available for instrumentation. To account for this issue, In order to simulate the monitored execution of API methods, we propose to associate each API method with four special JavaScript methods – *domain*, *check*, *label*, and *taint* – that we call its *IFlow Signature.* Concretely: **(1)** *domain* checks whether or not to apply the API, **(2)** *check* checks whether the constraints associated with the API are verified, **(3)** *label* updates the instrumented labelling and outputs the reading effect associated with a call to the API, and **(4)** *taint* marks the objects generated by the API so that they are not confused with normal JavaScript objects. Observe that functions *check* and *label* must be specified separately because *check* has to be executed before

Property Look-up

$$\frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0 \mid j\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1 \mid k\rangle \qquad \mathcal{C}\langle e_0[e_1]^i\rangle = \langle \hat{e}_0, \hat{e}_1, \hat{e} \mid i\rangle \\ e' = \lceil \mathit{\#tmp} = \$\mathit{register}(\$v_j, \$v_k), \mathit{\#tmp}\ ?\ e_{api} : \hat{e}\rceil \\ e_{api} = \lceil \mathit{\#tmp.check}(\$l_j, \$l_k, \$v_j, \$v_k), \$v_i = \$v_j[\$v_k], \mathit{\#tmp.label}(\$v_i, \$l_j, \$l_k, \$v_j, \$v_k)\rceil\end{array}}{\mathcal{C}_{api} e_0[e_1]^i = \langle \hat{e}_0, \hat{e}_1, e' \mid i\rangle}$$

Method Call

$$\frac{\begin{array}{c}\mathcal{C}\langle e_0\rangle = \langle \hat{e}_0 \mid j\rangle \qquad \mathcal{C}\langle e_1\rangle = \langle \hat{e}_1 \mid k\rangle \qquad \mathcal{C}\langle e_2\rangle = \langle \hat{e}_2 \mid l\rangle \qquad \mathcal{C}\langle e_0[e_1](e_2)^i\rangle = \langle \hat{e}_0, \hat{e}_1, \hat{e}_2, \hat{e} \mid i\rangle \\ e' = \lceil \mathit{\#tmp} = \$\mathit{register}(\$v_j, \$v_k), \mathit{\#tmp}\ ?\ e_{api} : \hat{e}\rceil \\ e_{api} = \lceil \mathit{\#tmp.check}(\$l_j, \$l_k, \$l_l, \$v_j, \$v_k, \$v_l), \$v_i = \$v_j[\$v_k](\$v_l), \mathit{\#tmp.label}(\$v_i, \$l_j, \$l_k, \$l_l, \$v_j, \$v_k, \$v_l)\rceil\end{array}}{\mathcal{C}_{api} e_0[e_1](e_2)^i = \langle \hat{e}_0, \hat{e}_1, \hat{e}_2, e' \mid i\rangle}$$

Figure 6.5: Extended Compiler - $\mathcal{C}_{API}$

calling the API (in order to prevent its execution when it can potentially trigger a security violation), whereas *label* must be executed after calling the API (so that it can label the memory resulting from its execution). Formally, we define an *IFlow Signature* as a 4-tuple $\langle \mathit{\#check}, \mathit{\#label}, \mathit{\#domain}, \mathit{\#taint}\rangle$, where: *#check*, *#label*, *#domain*, and #taint are the references of the function objects corresponding to the *check*, *label*, *domain*, and *taint* functions, respectively.

We require the existence of a runtime function that simulates the API Register, which we denote by $\$Register$. The function $\$Register$ makes use of the methods *domain* of each API in its range to decide whether the current method call or property look-up is going to trigger the invocation of an external API, in which case it returns an object containing the corresponding IFlow Signature (otherwise it simply returns `null`). Figure 6.5 presents the extension of the inlining compiler introduced in Chapter 4 that takes into account the possible invocation of external APIs. We denote the new compiler by $\mathcal{C}_{API}$. This compiler coincides with the previous one for every program construct with the exception of method calls and property look-ups, in which case it has to take into account the possible invocation of external APIs. For these two constructs, the code generated by the compiler proceeds as follows: **(1)** it executes the statements corresponding to the compilation of its subexpressions; **(2)** it checks, using the values of to the first two subexpressions, whether that property look-up or method call is associated with an IFlow Signature (using the $\$Register$ function); **(3 -** ***true*****)** if it is the case, it executes the *check* method of the corresponding IFlow Signature, an expression equivalent to the original expression, the *label* method in order to update the generated memory, and the *taint* method to register that the returned value was produced by an API call; **(3 -** false**)** if it is not, it executes the compilation of an expression equivalent to the compilation of the original one by the original inlining compiler.

6.3.1 IFlow Signatures for the Queue API

6.4 Related Work

6.4.1 Security of Web APIs

Taly et al. [Taly 2011] study the problem of API confinement. They provide a static analysis designed to formally verify whether, when integrating the code of an API in an arbitrary page, the integrator code cannot interact with the API and cause it to leak its confidential resources. They consider a lexically-scoped fragment of JavaScript in which property look-up and property update/create expressions are explicitly annotated with the set of properties that can be possibly read or written. They use, however, a more restrictive notion of a web API than the one used in this work, in the sense that they use the term API only to refer to JavaScript libraries whose code is explicitly included by the programmer and, therefore, available for either runtime or static analysis.

queue:

check:

```
function(lev0, lev1, val0, val1) { true }
```

label:

```
function(queue, lev0, lev1, val0, val1, lev_struct) {
   var qnab = {},
   qnlab.queue = _lat.lub(lev0, lev1),
   qnlab.struct = _lat.lub(lev0, lev1, lev_struct),
   qnlab.qnodes = [],
   queueAPI.registerQueueLab(queue, qnlab),
   _lat.lub(lev0, lev1)
}
```

domain:

```
function(val0, val1) { (val0 == queueAPI) && (val1 == 'queue') }
```

pop:

check:

```
function(lev0, lev1, val0, val1) {
   var qnlab, constraint_lev, ctxt_lev, last_lev;
   qnlab = queueAPI.getQueueLab(val0),
   ctxt_lev = _lat.lub(lev0, lev1),
   last_lev = qnlab.qnodes[qnlab.length - 1],
   constraint_lev = _lat.glb(qnlab.struct, last_lev),
   _lat.lt(ctxt_lev, constraint_lev) ? true : _diverge()
}
```

label:

```
function(val_ret, lev0, lev1, val0, val1) {
   var qnlab;
   qnlab = queueAPI.getQueueLab(val0),
   qnlab.qnodes.pop()
}
```

domain:

```
function(val0, val1) { isQueue(val0) && (val1 == 'pop') }
```

Table 6.1: IFlow Signatures For the Queue API - queue and pop

push:

check:

```
function(lev0, lev1, lev2, lev3, val0, val1, val2, val3) {
   var queue_lab, ctxt_lev, cnstr_0, cnstr_1, new_pos;
   queue_lab = queueAPI.getQueueLab(val0),
   cnstr_0 = _lat.lt(_lat.lub(lev0, lev1), queue_lab.struct),
   cnstr_1 = _lat.lt()
   constraint_lev = _lat.lub(, queue_lab.qnodes),
   _lat.lt(ctxt_lev, constraint_lev) ? true : _diverge()
}
```

label:

```
function(val_ret, lev0, lev1, val0, val1) {
   var queue_lab;
   queue_lab = getQueueLab(val0),
   queue_lab.qnodes.pop()
}
```

domain:

```
function(val0, val1) { isQueue(val0) && (val1 == 'pop') }
```

Table 6.2: IFlow Signatures For the Queue API - push

Chapter 7

Monitoring Secure Information Flow in a DOM-like API

Contents

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [Recommendation 2005]. In contrast to the ECMA Standard [5th edition of ECMA 262 June 2011 2011] that specifies in full detail the internals of objects created during the execution, the DOM API only specifies the behaviour that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine but by a separate engine whose role is to do so, often called the *render engine*. Therefore, the design of an information flow monitor for client-side JavaScript Web applications must take into account the DOM API.

Russo et al. [Russo 2009] first studied the problem of information flow control in dynamic tree structures, for a model where programs are assumed to operate on a single current working node. However, in real client-side JavaScript, tree nodes are first-class values, which means that a program can store in memory several references to different nodes in the DOM forest at the same time. We present a set of monitor extensions for the extensible Core JavaScript monitor in order for it to take into account a fragment of the DOM Core Level 1 API, that we call Core DOM. In Core DOM, tree nodes are treated as first-class values and thus they support all operations available to other types of values, such as assignment to variables. Interestingly, this language design feature enables us to implement a more fine-grained information flow control mechanism, since it becomes possible to distinguish the security level of the node itself from both the security level of the value that is stored in the node and from the level of its position in the DOM forest. We prove that the proposed monitor extensions are noninterferent and therefore the extension of the Core JavaScript monitor with the Core DOM API is also noninterferent.

Live collections are a special kind of data structure featured in the DOM Core Level 1 API that automatically reflect the changes that occur in the document. There are several types of live collections.

For instance, the method `getElementsByTagName` returns a live collection containing the DOM nodes that match a given *tag name*. In the following example, after retrieving the initial collection of **(DIV)** nodes, the program iterates over the *current* size of this collection, while introducing a new **(DIV)** node at each step:

```
divs = document.getElementsByTagName("DIV"); i = 0;
while(i <= divs.length){
  document.appendChild(document.createElement("DIV")); i++; }
```

Every time a new **(DIV)** node is inserted in the `document` (no matter where in its structure), it is also inserted in the live collection bound to `divs`. Due to the live update of the loop condition, if the initial `document` contains at least one **(DIV)** node, the program does not terminate.

Live collections can be exploited to encode new types of information leaks. Therefore, we include in the Core DOM API several methods that capture the behavior of `getElementsByTagName` in the DOM API. Furthermore, we demonstrate that these constructs effectively augment the observational power of an attacker and we show how to monitor their execution in order to preserver noninterference.

In the remainder of this chapter, we start by formally introducing the targeted Core DOM API (Section 7.1). We then discuss the challenges of controling information flow in the considered API and present the monitor extensions for the Core DOM API (Section 7.2). The scenario is then extended with live collections, for which we propose an additional set of monitor extensions that tackle newly introduced forms of information leaks (Section 7.3).

7.1 Core DOM

The DOM data structure can be viewed as a forest of DOM nodes containing a special tree corresponding to the document being displayed by the browser. Interestingly, most of the information flows that are specific to the DOM API have to do with dynamic tree operations, such as the creation, insertion, or removal of DOM nodes in or from the DOM forest. Hence, in Core DOM, we include the most relevant methods and properties of the DOM Core Level 1 API used for traversing and updating tree structures. Concretely, in Core DOM, every node has a type, called its *tag* (for instance, **DIV**) and can store a single value taken from $\mathcal{P}rim$. All the nodes in memory form a *forest*, meaning that every node has a possibly empty list of *children* and at most a single *parent*. A node with no parent is called an *orphan* node. Whenever a node has a parent, we define its *index* as the position it occupies in the list of children of its parent. The Core DOM API is assumed to be available via the global variable *document* and to expose the following methods and properties:

- `document.createElement(tag)`: creates a new element node with tag name `tag`;
- `node0.appendChild(node1)`: appends `node1` to the list of children of `node0` provided that `node1` is an orphan node;
- `node0.removeChild(node1)`: removes `node1` from the list of children of `node0` provided that `node1` is indeed a child of `node0`;
- `node[i]`: evaluates to the i+1th child of `node` provided that it has at least $i + 1$ children;
- `node.length`: evaluates to the number of children of `node`;
- `node.parentNode`: retrieves the parent of `node`, that is, the node through which it is accessed in the DOM forest;
- `node.value`: retrieves the value that is stored in `node`;
- `node.store(value)`: stores `value` inside `node`.

We depart from the specification in that in Core DOM the child nodes of a given DOM node are directly accessed through their parent instead of through a special object *childNodes*. Hence, instead of writing $div1.childNodes[i]$ to access the ith child of the DIV element bound to $div1$, we simply write $div1[i]$.

$$\textsc{New}\quad \frac{r = fresh_{DOM}(f, i) \qquad f' = f\,[r \mapsto \langle m, null, null, \varepsilon\rangle]}{\langle f, \_ :: \_ :: m\rangle^{(i,\sigma_0,\sigma_1,\sigma_2)}\ \mathsf{new}\ \langle f', r\rangle^{(r,\sigma_0,\sigma_1,\sigma_2)}}$$

$$\textsc{Append}\quad \frac{\begin{array}{c}\langle r', r\rangle \notin \mathcal{R}_{Ancestor}(f) \qquad f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \qquad f(r') = \langle m', v', null, \overrightarrow{r}'\rangle \\ f' = f\,[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} :: r'\rangle, r' \mapsto \langle m', v', r, \overrightarrow{r}'\rangle]\end{array}}{\langle f, r :: \_ :: r'\rangle\ \mathsf{append}\ \langle f', r'\rangle^{(r,r')}}$$

$$\textsc{Remove}\quad \frac{\begin{array}{c}f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \qquad f(r).\mathsf{children}(i) = r' \qquad f(r') = \langle m', v', r, \overrightarrow{r}'\rangle \\ f' = f\,[r \mapsto \langle m, v, \hat{r}, Shift_L(\overrightarrow{r}, i)\rangle, r' \mapsto \langle m', v', null, \overrightarrow{r}'\rangle]\end{array}}{\langle f, r :: \_ :: r'\rangle\ \mathsf{remove}\ \langle f', r'\rangle^{(r,r')}}$$

$$\textsc{Index}\quad \frac{f(r).\mathsf{children}(i) = r'}{\langle f, r :: i\rangle\ \mathsf{index}\ \langle f, r'\rangle^{(r,r')}} \qquad \textsc{Length}\quad \frac{i = |f(r).\mathsf{children}|}{\langle f, r :: \_\rangle\ \mathsf{length}\ \langle f, i\rangle^{(r)}} \qquad \textsc{Parent}\quad \frac{f(r).\mathsf{parent} = v}{\langle f, r :: \_\rangle\ \mathsf{parent}\ \langle f, v\rangle^{(r)}}$$

$$\textsc{Value}\quad \frac{f(r).\mathsf{value} = v}{\langle f, r :: \_\rangle\ \mathsf{value}\ \langle f, v\rangle^{(r)}} \qquad \textsc{Store}\quad \frac{\begin{array}{c}f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \\ f' = f\,[r \mapsto \langle m, v', \hat{r}, \overrightarrow{r}\rangle]\end{array}}{\langle f, r :: \_ :: v'\rangle\ \mathsf{store}\ \langle f', v'\rangle^{(r)}}$$

Figure 7.1: Core DOM API - Semantics

7.1.1 Formal Semantics

We model a DOM forest $f : \mathcal{R}ef_{DOM} \rightarrow \mathcal{N}$ as a partial mapping from a set of DOM references to the set of DOM nodes. A DOM node is a tuple of the form: $\langle m, v, r, \overrightarrow{r}\rangle$, where: **(1)** m is the node's tag, **(2)** v the value it stores, **(3)** r the reference pointing to its parent, and **(4)** $\overrightarrow{r}$ its list of children (more precisely, a list of references, each pointing to one of its children). For simplicity, given a DOM node n, we denote by $n.\mathsf{tag}$, $n.\mathsf{value}$, $n.\mathsf{parent}$, and $n.\mathsf{children}$ its tag, value, parent, and list of children, respectively. The semantics of Core DOM makes use of a semantic function $\mathcal{R}_{Ancestor}$ that, given a forest f, outputs a binary relation in $\mathcal{R}ef_{DOM} \times \mathcal{R}ef_{DOM}$ such that $\langle r_0, r_1\rangle \in \mathcal{R}_{Ancestor}(f)$ iff the node pointed to by r_0 is an ancestor of that pointed to by r_1. Figure 7.1 presents the formal specification of the methods that compose the Core DOM API. In the following, we use $Shift_L(L, i)$ for the list obtained by removing from L its i^{th} element (provided that it is defined).

As happened with the Queue API, the formal semantics of the Core DOM API assumes that the references used by this API do not overlap with the references used by the standard semantics, meaning that the co-domain of the allocator used in its specification, $fresh_{DOM}$, is assumed not to overlap with that of the allocator of standard Core JavaScript. In other words, $\mathcal{R}ef_{DOM}$ does not overlap with $\mathcal{R}ef$. Furthermore, we assume that every memory contains a special object called *document* accessible through the property *document* of the global object and stored in a fixed reference $\#doc\mathcal{R}ef_{DOM}$. The API register is given below:

$$\mathcal{R}_{DOM}(v_0, v_1, m) = \begin{cases} \mathsf{new} & \text{if } v_0 = \#docv_1 = \text{“createElement”} \wedge m = \text{“MC”} \\ \mathsf{append} & \text{if } v_1 = \text{“appendChild”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“MC”} \\ \mathsf{remove} & \text{if } v_1 = \text{“removeChild”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“MC”} \\ \mathsf{index} & \text{if } v_1 \in \mathcal{N}um \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“LU”} \\ \mathsf{length} & \text{if } v_1 = \text{“length”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“LU”} \\ \mathsf{parent} & \text{if } v_1 = \text{“parentNode”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“LU”} \\ \mathsf{value} & \text{if } v_1 = \text{“nodeValue”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“LU”} \\ \mathsf{store} & \text{if } v_1 = \text{“storeValue”} \wedge v_0 \in \mathcal{R}ef_{DOM} \wedge m = \text{“MC”} \end{cases}$$

7.2 Monitor Extensions for Core DOM

Before proceeding to describe the monitor for securing information flow in Core DOM, we discuss the main challenges imposed by the particular features of this API and how we propose to tackle them.

7.2.1 Challenges for Information Flow Control in Core DOM

The range of tree operations offered by Core DOM allows information to be stored and inspected from arbitrary nodes in several ways: **(1)** A node can be created and its existence tested; **(2)** A value can be stored and read from a node; **(3)** A node can be inserted at/removed from a certain position and both the number of children its former/new parent as well as the new positions of its former/new right siblings can be retrieved, where the *position* of a node can be understood as the pair consisting of its parent in the DOM forest and its index. These operations can be used to encode security leaks via the different information components that are associated with every node. We now examine these leaks and introduce the formal techniques we use for tackling them. In the examples, we assume that the original memory contains three initial **DIV** nodes, bound to `div0`, `div1`, and `div2` respectively and created as follows:

```
div0 = document.createElement("DIV"),
div1 = document.createElement("DIV"),
div2 = document.createElement("DIV")
```

7.2.1.1 Differentiating Information Components

Each node in a DOM forest can be seen to carry four main information components: its existence, its value, its position and its number of children. To some degree, these components can be manipulated separately, and there is value in treating them individually by the security analysis. For instance, in the following program, the final position of `div2` carries *high* information (because it is inserted in a *high* context), despite the fact that it contains the *low* level value originally stored in `l0`.

```
div2.storeValue(l0),
h ? div0.appendChild(div2) : div1.appendChild(div2)
```

After the execution of this program, the position of `div2` should not be revealed to a low observer. Its value, however, can be made public. Hence, while the evaluation of `div2.parentNode` should yield a high value, the evaluation of the `div2.nodeValue` in the final memory can yield a low value. Similarly, there is no reason why the position of a node that stores a secret value should not be public.

By treating tree nodes as first-class values, we can naturally differentiate the security levels that are associated to each of the node's information components. We propose to associate every tree node with four security levels. The *value level* of a node is the level of the value that it stores. The *position level* of a node is the level of its position in the DOM forest. Hence, the position level of a node constitutes an upper bound on the levels of the contexts in which its position in the DOM forest can change (such as by its insertion/removal). In other words, if a node has a *low* position level, it cannot be inserted in a new position or removed from its current one in a *high* context. The *structure security level* of a node is associated to the node's number of children. It serves as an upper bound on the levels of the contexts in which the number of children of a node can be changed (such as by insertion/removal of nodes in/from its list of children). Finally, the *node level* is the level associated to information about the existence of the node itself. It is used as an upper bound on the levels of the contexts in which the node can be created or a lower bound on its own value and structure level, and on its children's position levels.

7.2.1.2 Security Leaks in Core DOM

When removing a node from the list of children of a given node, the indexes of its right siblings change, thereby entailing a new kind of implicit flow. Consider the following example

```
div0.appendChild(div1),
div0.appendChild(div2),
h ? (div0.removeChild(div1)) : (null),
l0 = div0[0]
```

Program:	$h = 0$	$h = 1$	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
$l = \texttt{tt}$	$\Gamma(l) := L$	$\Gamma(l) := L$	$\Gamma(l) := L$
h ?	branch not taken	branch taken	branch taken
$div0.appendChild(div1)$	—	$\Sigma(r_0).\mathsf{struct} := H$	*stuck*
$(div0.length == 0)$?	branch taken	branch not taken	—
$l = \texttt{ff}$	$\Gamma(l) := L$	—	—
Final Low Memory:	$l = \texttt{ff}$	$l = \texttt{tt}$	—

Table 7.1: The Structure Security Level of DOM Nodes Must Be Flow Insensitive

that serves as the running example in this subsection. This program prepends `div1` and `div2` to the list of children of `div0` (which is originally empty). Then, depending on the value of the *high* variable `h`, it removes `div1` from the list of children of `div0`. Hence, depending on the value of `h`, the program assigns either `div1` or `div2` to the *low* variable `l0`. We refer to these forms of security leaks as *order leaks*, as they leverage information about the order of the nodes in the list of children of their parents.

In a nutshell, when removing one node from the list of children of another, the positions of its right siblings also change. Complementarily, when appending a node to the list of children of another, the position it occupies depends on the positions of its left siblings. Therefore, the monitor enforces the position levels of the right siblings of a given node to be equal to or higher than its own position level. Suppose, for instance, that: **(1)** a node B is removed from the list of children of a node A in an invisible context and that **(2)** that B has a right sibling C. For the monitor to allow this removal to go through, B has to have an invisible position level, since the position of B can only change in a context whose level is lower than or equal to its position level. Furthermore, since the position level of C is higher than or equal to the position level of B, we conclude that C must also have an invisible position level. Hence, the removal of B does not cause any visible changes.

When moving from one node to another, information about the position of the child node is leaked. For instance, in the program above the evaluation of `div0`[0] leaks information about the position of the first child of `div0`. Concretely, we get to know its index and its parent node. Since in this example such information depends on the value of the *high* variable h, we conclude that the evaluation of `div0`[0] leaks information at level H.

The fact that a program can inspect the number of children of a given node can be exploited to encode implicit information flows. If we add the low assignment `l1 = div0.length` to the end of the program above, the value of `l1` will be set to 2 or to 1 depending on the value of the *high* variable h. The *structure security level* of a DOM node is meant to control this kind of leaks. One can look at the *structure security level* of a DOM node as an upper bound on the levels of the contexts in which one can add or remove nodes to or from its list of children. Hence, if a node has *low* structure security level, one cannot insert/remove nodes in/from its list of children in *high* contexts. Therefore, the level associated with looking-up the number of children of a given node corresponds to its structure security level. For instance, for the program above to be legal, the structure security level of `div0` must be H. Hence, the level associated with the evaluation of `div0.length` is H independently of the original value of h.

7.2.1.3 Flow-sensitive versus Flow-insensitive Monitoring in Core DOM

Both the structure security level and the position level of a node are used to control the implicit flows that can be encoded by inserting/removing nodes in/from the DOM forest. Hence, in order to apply the no-sensitive-upgrade discipline, these levels cannot be upgraded. This point is illustrated in Table 7.1, which represents four monitored executions of a program (represented on the left) in two distinct memories, by showing how the Core JavaScript labelling Σ as well as the Core DOM API labelling Ξ evolve during each execution. The initial memories are such that div_0 and div_1 each bind an orphan node with *low* structure security level, and are pointed to by r_0 and r_1, respectively, but differ in the value of *high* variable h.

While the monitor following the *naive approach* raises the structure security level of div_0 to H (allowing the execution to go through), the monitor following the *no-sensitive-upgrade* discipline blocks the execution when the program tries to append div_1 to the list of children of div_0 in a *high* context. The case regarding the position level can be seen by replacing the test of the second conditional expression with $div_1.parentNode$, assuming that the original position level of div_1 is *low*. In contrast to the position level and to the structure security level, the value level of a node can be upgraded, as the value stored in a node is set explicitly. However, such upgrades cannot be caused by implicit information flows.

7.2.2 An Attacker Model for the Core DOM API

In order to formally characterise what part of a DOM forest an attacker at a given security level can observe, we must define a low-equality relation $\sim_{DOM}$ for DOM forests parameterizable in an arbitrary security level σ. Two forests f and f' respectively labeled by Ξ and Ξ' are related by $\sim^{\sigma}_{DOM}$ if an attacker at level σ cannot distinguish the two of them. To this end, we define a node labeling $\Xi : \mathcal{R}ef_{DOM} \to \mathcal{L}^4$ as a function that associates each DOM reference with a tuple of four security levels. Hence, given a DOM reference r and a labeling Ξ, $\Xi(r) = \langle \sigma_n, \sigma_v, \sigma_p, \sigma_s \rangle$, where: **(1)** σ_n is the node level, **(2)** σ_v is the value level, **(3)** σ_p is the position level, and **(4)** σ_s is the structure security level. For clarity, given a node n pointed to by a reference r and a node labelling Ξ, we denote by $\Xi(r).\mathsf{node}$, $\Xi(r).\mathsf{value}$, $\Xi(r).\mathsf{pos}$, and $\Xi(r).\mathsf{struct}$ its node level, value level, position level, and structure security level, respectively. For simplicity, we impose four restrictions on the levels assigned to a given node. First, one cannot store a visible value in an invisible node. Second, an invisible node cannot have a visible position. Third, an invisible node cannot have a visible number of children. Fourth, an invisible node cannot have a visible node in its list of children (in practice, this means that we cannot insert a visible node in an invisible node). Formally, for every reference $r \in dom(()\Xi)$, it holds that: $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r).\mathsf{value} \sqcap \Xi(r).\mathsf{pos} \sqcap \Xi(r).\mathsf{struct}$. Additionally, for every two DOM references r and r' in a forest f such that: $r, r' \in dom(\Xi)$ and $f(r).\mathsf{children}(i) = r'$ for some integer i, it holds that $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r').\mathsf{node}$.

The low-project of a DOM forest f labeled by Ξ at a given security level σ, formally given in Definition 7.1, establishes that an attacker at level σ can see: **(1)** the DOM references whose corresponding nodes are associated with levels $\sqsubseteq \sigma$ as well as their tags, **(2)** the values stored in visible nodes whose value level is $\sqsubseteq \sigma$, **(3)** the positions of visible nodes (with visible parents) whose levels are $\sqsubseteq \sigma$, and **(4)** the number of children of visible nodes whose structure security level is $\sqsubseteq \sigma$.

Definition 7.1 (Low-Projection and Low-Equality for DOM Forests). *The low-projection of a forest f w.r.t. a security level σ and a labeling Ξ is given by:*

$$\begin{aligned} f \upharpoonright^{\Xi,\sigma} = & \ \{(r, f(r).\mathsf{tag}, \Xi(r).\mathsf{node}, \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct}) \mid \Xi(r).\mathsf{node} \sqsubseteq \sigma\} \\ \cup & \ \{(r, f(r).\mathsf{value}, \Xi(r).\mathsf{value}) \mid \ \Xi(r).\mathsf{value} \sqsubseteq \sigma\} \\ \cup & \ \{(r, i, r') \mid \ f(r).\mathsf{children}(i) = r' \ \wedge \ \Xi(r').\mathsf{pos} \sqsubseteq \sigma\} \\ \cup & \ \{(r, null) \mid \ f(r).\mathsf{parent} = null \ \wedge \ \Xi(r).\mathsf{pos} \sqsubseteq \sigma\} \\ \cup & \ \{(r, |f(r).\mathsf{children}|) \mid \ \Xi(r).\mathsf{struct} \sqsubseteq \sigma\} \end{aligned}$$

Two forests f_0 and f_1, respectively labeled by Ξ_0 and Ξ_1 are said to be low-equal at security level σ, written $f_0, \Xi_0 \sim^{\sigma}_{DOM} f_1, \Xi_1$, if they coincide in their respective low-projections, meaning that $f_0 \upharpoonright^{\Xi_0,\sigma} = f_1 \upharpoonright^{\Xi_1,\sigma}$.

Table 7.2 represents the final forests obtained from the execution of the program given in the beginning of the previous subsection in two distinct memories that initially map the *high* variable h to 1 and to 0, respectively. On the left, we represent each of the final forests, whereas on the right, we represent their coinciding low-projection. Nodes are labeled with their node level and structure security level, while edges are labeled with the child's position level. The position levels of div_1 and div_2 as well as the structure security level of div_0 are assumed to be originally *high*. All other levels are assumed to be originally *low*.

7.2.3 Enforcement

Figure 7.2 presents the monitor methods for the Core DOM API. Let us briefly explain the rules of the proposed monitor extensions.

The Rule [New] expects its input to be annotated with the DOM reference in which the new DOM node is to be allocated as well as its node level, position level, and structure security level. This rule checks whether the node level is higher than or equal to the level of the resources that were use to decide

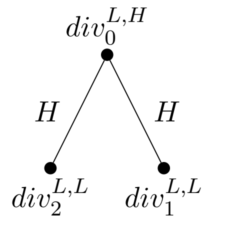
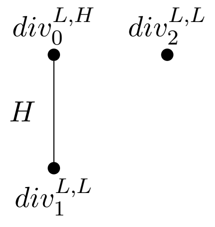
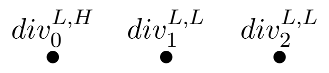
Final Memory		*Final Memory Low-Projection*
$h = 0$	$h = 1$	Both $h = 0$ and $h = 1$
		

Table 7.2: Two DOM forests and Their Low Projections

the application of this API. Moreover, it checks that the node level of the node to be created is lower than or equal to its position level and structure security level.

The Rules [Append] and [Remove] prevent the removal/insertion of a node with a visible position in an invisible context as well as the alteration of the number of children of a node with a visible number of children in an invisible context. Moreover, since the position that a node occupies in the list of children of its parent after being appended depends on the positions of its left siblings, the Rule [Append] also checks that the position level of the node being inserted is higher than or equal to the position level of its left siblings. Hence, this rule ensures that the position levels of the children of every DOM node are always monotonically increasing.

The Rules [Index] and [Parent] do not change the DOM forest. Hence, they are not subject to any constraint. The reading effect of these rules is the least upper bound of the levels of resources that were used to decide their application (σ_0 and σ_1) and the level of the arriving or departing node's position, respectively. Going from a parent node to one of its children using the `index` API leaks information about the position of that particular child. Likewise, going from a child node to its parent using the `parent` API leaks information about the position of the child and not that of the parent. Hence, in both rules, only the position level of the child node is included in the reading effect of the API call. Finally, the levels of the nodes that the traversed edge connects are ignored, as they are enforced to be lower than or equal to the child's position level.

As the use of the `length` API about the number of children of the node on which it is called, the Rule [Length] includes in its computed reading effect the levels of resources that were used to decide its application (σ_0 and σ_1) as well as the structure security level of the node's on which it is applied. Since it does not change the DOM forest, it is not subject to any constraint.

The reading effect of the Rule [Value] is simply the least upper bound of the levels of the resources that were used to decide its application (σ_0 and σ_1) and the value level of the node whose value is being inspected. In order to prevent sensitive upgrades, the Rule [Store] checks whether the current value level of the node whose value is being updated is higher than or equal to the level of the context. Hence, updates of visible values in invisible contexts cause the execution of this API to abort.

7.3 Secure Information Flow for Live Collections

The DOM API includes several methods that return live collections. For instance, the method `getElementsByTagName` returns a live collection containing all the nodes in the document tree whose tag matches the string given as input. The distinctive feature of live collections is that they automatically reflect modifications to the document. Hence, every time a node matching the query that generated a given live collection is inserted/removed in/from the document, it is also automatically inserted/removed in/from that live collection. Therefore, rather than a simple static data structure, a live collection is in fact a dynamic query to the document.

The nodes of a live collection are arranged in *document order.* According to the specification, the order of a node is determined by the position in which "the first character of [its] XML representation occurs in the XML representation of the document after expansion of general entities" [Recommendation 2005]. In other words, the document order is an ordering $\leq$ on the nodes of the DOM forest such that for every

$$\textsc{New}\quad \frac{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_n \sqsubseteq \sigma_p \sqcap \sigma_s \qquad \Xi' = \Xi\left[r \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s\rangle\right]}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}\ \mathsf{new}_{lab}\ \langle \Xi', \sigma_0\rangle}$$

$$\textsc{Append}\quad \frac{\begin{array}{c} r'' = null \vee \Xi(r'').\mathsf{pos} \sqsubseteq \Xi(r').\mathsf{pos} \qquad \Xi(r).\mathsf{node} \sqsubseteq \Xi(r').\mathsf{node} \\ \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos}\end{array}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{(r,r',r'')}\ \mathsf{append}_{lab}\ \langle \Xi, \Xi(r').\mathsf{pos}\rangle}$$

$$\textsc{Remove}\quad \frac{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Sigma(r).\mathsf{struct} \sqcap \Sigma(r').\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{(r,r')}\ \mathsf{remove}_{lab}\ \langle \Xi, \Sigma(r').\mathsf{pos}\rangle} \qquad \textsc{Index}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r').\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r,r')}\ \mathsf{index}_{lab}\ \langle \Xi, \sigma\rangle}$$

$$\textsc{Length}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{struct}}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r)}\ \mathsf{length}_{lab}\ \langle \Xi, \sigma\rangle} \qquad \textsc{Parent}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r').\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r,r')}\ \mathsf{parent}_{lab}\ \langle \Xi, \sigma\rangle}$$

$$\textsc{Value}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{value}}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r)}\ \mathsf{value}_{lab}\ \langle \Xi, \sigma\rangle} \qquad \textsc{Store}\quad \frac{\begin{array}{c}\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Sigma(r).\mathsf{node} \qquad \sigma_0 \sqcup \sigma_1 \sqsubseteq \Xi(r).\mathsf{value} \\ \Xi' = \Xi\left[r \mapsto \langle \Xi(r).\mathsf{node}, \sigma, \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct}\rangle\right]\end{array}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{(r)}\ \mathsf{store}_{lab}\ \langle \Xi', \sigma\rangle}$$

Figure 7.2: Core DOM Monitor - Primitives for Tree Operations

two nodes n_0 and n_1 in the same DOM tree, $n_0 \leq n_1$ if and only if n_0 is found before n_1 in a **depth-first left-to-right** search starting from the root of that tree.

In this section we extend the Core DOM API with the following methods and properties for handling live collections:

- `node.getElementsByTagName(tagName)`: creates a new live collection containing all the descendants of the node `n` with tag name `tagName` in document order;
- `lc[i]`: retrieves the i+1th node in the live collection `lc`;
- `lc.length`: returns the number of nodes in the live collection `lc`.

7.3.1 A Semantics for Live Collections

In modelling the semantics of live collections, we chose to re-compute the content of a live collection every time a program tries to look-up one of its elements or its number of elements. The alternative approach would be to compute it only once and, every time there were changes in the document's structure, to reflect those changes in all existing live collections. This second approach has, among others, the disadvantage of scattering the semantics of live collections through all the DOM API methods that modify the structure of the document. Hence, in order to model live collections, we extend our model of the DOM with a new type of data structure, called a *live collection*, taken from a set $\mathcal{L}ives$. We model a live collection as a tuple of the form $\langle r, m\rangle$, where r is the reference of the DOM node on which the query was issued and m the corresponding query. For instance, the evaluation of `div0.getElementsByTagName("DIV")` generates a live collection containing the reference of the node bound to `div0` and the string `"DIV"`. For simplicity, we assume that live collections are allocated in a set of references, $\mathcal{R}ef_{live}$, that does not overlap with neither the one used for the allocation of Core JavaScript objects nor the one used for the allocation of DOM nodes. Accordingly, we redefine a DOM API state ν as a pair $\langle f, lives\rangle$ consisting of a DOM forest f and a partial function $lives : \mathcal{R}ef_{live} \to \mathcal{L}ives$, which we call *live collection register*, mapping live collection references to live collections. Given a DOM API state ν, we denote by $\nu.f$ and $\nu.lives$ its corresponding DOM forest and live collection register, respectively.

$$\textsc{Live New}\quad \frac{r' = fresh_{live}(i, \nu.lives) \qquad lives' = \nu.lives\,[r' \mapsto \langle r, m\rangle]}{\langle \nu, r :: \_ :: m\rangle^{i}\ \mathsf{newLive}\ \langle\langle \nu.f, lives'\rangle, r'\rangle^{(r')}}$$

$$\textsc{Live Length}\quad \frac{\nu.lives(r) = \langle r', m\rangle \qquad \nu.f \vdash r' \rightsquigarrow_m \overrightarrow{r}}{\langle \nu, r :: \_\rangle\ \mathsf{liveLength}\ \langle \nu, |\overrightarrow{r}|\rangle^{(r,m,\nu.f)}}$$

$$\textsc{Live Item}\quad \frac{\nu.lives(r) = \langle r', m\rangle \qquad \nu.f \vdash r' \rightsquigarrow_m \overrightarrow{r} \qquad \overrightarrow{r}(i) = r''}{\langle \nu, r :: i\rangle\ \mathsf{liveIndex}\ \langle \nu, r''\rangle^{(r,r',\nu.f)}}$$

Figure 7.3: Monitor Extensions for Handling Live Collections

$$\textsc{Node Not Found - Orphan Node}\quad \frac{|f(r).\mathsf{children}| = 0 \qquad f(r).\mathsf{tag} \neq m}{f \vdash r \rightsquigarrow_m \varepsilon}$$

$$\textsc{Node Not Found - Non-Orphan Node}\quad \frac{\begin{array}{c}\overrightarrow{r} = f(r).\mathsf{children} \qquad |\overrightarrow{r}| = n \qquad f(r).\mathsf{tag} \neq m \\ \forall_{0 \leq i < n}\ f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i\end{array}}{f \vdash r \rightsquigarrow_m \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}}$$

$$\textsc{Node Found - Orphan Node}\quad \frac{|f(r).\mathsf{children}| = 0 \qquad f(r).\mathsf{tag} = m}{f \vdash r \rightsquigarrow_m r :: \varepsilon}$$

$$\textsc{Node Found - Non-Orphan Node}\quad \frac{\begin{array}{c}\omega = f(r).\mathsf{children} \qquad |\overrightarrow{r}| = n \qquad f(r).\mathsf{tag} = m \\ \forall_{0 \leq i < n}\ f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i\end{array}}{f \vdash r \rightsquigarrow_m r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}}$$

Figure 7.4: Search Predicate

Figure 7.3 gives the formal specification of the methods of the DOM API for handling live collections. The semantics makes use of a parametric allocator $fresh_{live}$ and a search predicate of the form $f \vdash r \rightsquigarrow_m \overrightarrow{r}$, formally given in Figure 7.4, which formalizes the search for the nodes matching a given tag in a tree. Intuitively, given a forest f, a node reference r, a tag name m, and a list of node references $\overrightarrow{r}$, $f \vdash r \rightsquigarrow_m \overrightarrow{r}$ holds iff $\overrightarrow{r}$ is the list of all the nodes with tag m found when traversing the tree of f rooted at r in **document order**. Finally, the API register is given below:

$$\mathcal{R}_{Live}(v_0, v_1, m) = \begin{cases} \mathsf{newLive} & \text{if } v_0 \in \mathcal{R}ef_{DOM} \wedge v_1 = \text{“getElementsByTagName”} \wedge m = \text{“MC”} \\ \mathsf{liveIndex} & \text{if } v_0 \in \mathcal{R}ef_{Live} \wedge v_1 \in \mathcal{N}um \wedge m = \text{“LU”} \\ \mathsf{liveLength} & \text{if } v_0 \in \mathcal{R}ef_{Live} \wedge v_1 = \text{“length”} \wedge m = \text{“LU”} \end{cases}$$

7.3.2 Information Leaks introduced by Live Collections

Live collections can be exploited to encode new types of information leaks. We now discuss discuss the main challenges imposed by the introduction of live collections as well as how we propose to tackle them.

7.3.2.1 Leaks via the Size of Live Collections

Consider the program below, which is to be executed in a forest that originally contains five orphan **DIV** nodes respectively bound to the variables `div0`, `div1`, `div2`, `div3`, and `div4`.

```
div0.appendChild(div1),
div0.appendChild(div2),
div0.appendChild(div3),
lc0 = div0.getElementsByTagName("DIV"),
h ? (div1.appendChild(div4)) : (null),
l0 = lc0.length
```

Depending on the value of `h`, `l` may be either set to 4 or to 5. In order to tackle this type of leak, we require the programmer to pre-establish for each possible tag name m an upper bound on the position

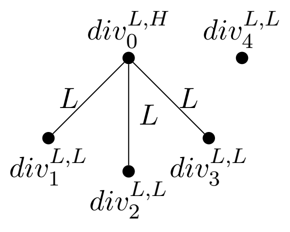
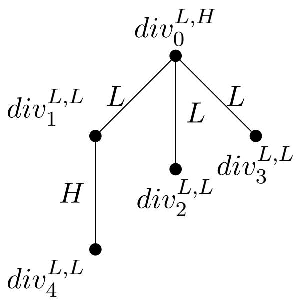
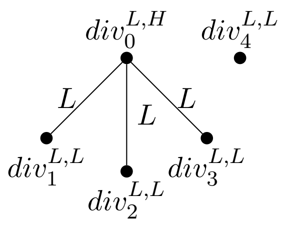
Final Memory		*Final Memory Low-Projection*
$h = 0$	$h = 1$	Both $h = 0$ and $h = 1$
 $div_0^{L,H}$ $div_4^{L,L}$ L L L $div_1^{L,L}$ $div_2^{L,L}$ $div_3^{L,L}$	 $div_0^{L,H}$ $div_1^{L,L}$ L L L $div_3^{L,L}$ H $div_2^{L,L}$ $div_4^{L,L}$	 $div_0^{L,H}$ $div_4^{L,L}$ L L L $div_1^{L,L}$ $div_2^{L,L}$ $div_3^{L,L}$

Table 7.3: Two DOM Forests and their Low Projections

levels of the nodes with that tag name, which we denote by σ_m and call *global position level.* For instance, $\sigma_{\textbf{DIV}}$ corresponds to the pre-established upper bound on the position levels of **DIV** nodes. In the evaluation of `lc0.length`, the monitor first checks whether the position levels of all **DIV** nodes in the DOM forest are lower than or equal to the global position level. If that is the case, the execution is allowed to go through and the level associated with `lc0.length` is $\sigma_{\textbf{DIV}}$. Otherwise, the execution is aborted. Therefore, for this program to be legal $\sigma_{\textbf{DIV}}$ must be set to H, which means that the evaluation of `lc0.length` yields a value of level H.

The *global position level* is used to control the implicit flows that can be encoded via the inspection of the `length` property of live collections. Hence, it cannot be flow-sensitive, since upgrading the global position level constitutes, by definition, a **sensitive upgrade**.

7.3.2.2 Leaks via the Inspection of Live Collections

The inspection of an element of a live collection leverages information about the position it occupies in that live collection and therefore in the document structure. Hence, live collections introduce a new type of *order leak.* Consider, for instance, the following program:

```
div0.appendChild(div1),
div0.appendChild(div2),
div0.appendChild(div3),
lc0 = div0.getElementsByTagName("DIV"),
h ? (div1.appendChild(div4)) : (null),
l0 = lc0[3]
```

Here, depending on the value of the *high* variable `h`, `l0` is assigned either to `div3` or to `div2`. Hence, the API monitor must be able to detect this information flow and signal that the evaluation of `lc0`[3] leaks information at level H. Let us ignore by now the information flows triggered by the DOM operations involving live collections. According to the current enforcement mechanism, for this program to be legal the position level of `div4` as well as the structure security level of `div1` must be *high.* All other labels may be set to L. Table 7.3 represents on the left the final forests obtained from the execution of this program in two distinct memories that initially map the `h` to 0 and to 1, respectively. On the right, it represents their (coinciding) low-projection. In spite of being evaluated in two low-equal forests and of only handling visible values, the evaluation of `lc0`[3] yields two different values.

This example clearly shows that the use of live collections enhances the observational power of an attacker. This happens because live collections allow an attacker to operate on the nodes with the same tag in the same tree as if they were siblings. Hence, it is necessary to adjust the notion of a node's position in order to take into account this new way of traversing the DOM forest. Let the *live index* of a node be its position in the list of nodes obtained by searching its corresponding tree for the nodes with its tag in document order (where by its corresponding tree we mean the largest tree that includes it). The position of a node must now be understood as the triple consisting of its parent, its index, and its live index. Hence, changing the position of a node in a tree causes the positions of the nodes with the same tag in the same tree with higher live indexes to change. In order to deal with this kind of flow, the proposed enforcement mechanism guarantees that one can only inspect a live collection if the position levels of the nodes it "contains" monotonically increase in **document order**. For instance, in Table 7.3, the final forest obtained when $h = 1$ does not comply with this requirement because the position level of `div4` is not lower than or equal to the position level of `div2`, while the live index of `div4` is lower than the live index of `div2`.

7.3.3 An Attacker Model for Live Collections

At the formal level, the introduction of live collections poses an important challenge: how to model the enhanced observational power of an attacker that can use live collections to inspect the DOM forest? To answer this question formally means: **(1)** restating the low-equality definition for forests so as to correctly capture the observational power of such an attacker and **(2)** introducing a new low-equality for live collection registers. In order to do these, we have to extend the notion of DOM labelling to take into account live collection registers. Hence, an extended DOM labelling Ξ must now be modelled as pair $\langle \Xi_0, \Xi_1 \rangle$, where Ξ_0 is the *forest labelling* as defined in the previous section and $\Xi_1 : \mathcal{R}ef_{live} \rightarrow \mathcal{L}$ is the *live collection register labelling.* Concretely, given a live collection reference r, if $\Xi_1(r) = \sigma$, then the existence of the live collection pointed to by r is only visible at levels higher than or equal to σ. For simplicity, given a DOM labelling $\Xi = \langle \Xi_0, \Xi_1 \rangle$, we denote Ξ_0 and Ξ_1 respectively by $\Xi.f$ and $\Xi.lives$.

The low-equality for live collection registers is given in Definition 7.2. As mentioned above, this definition simply states that an attacker at level σ can only see the existence of live collections labelled with levels $\sqsubseteq \sigma$.

Definition 7.2 (Low-Projection and Low-Equality for Live Collection Registers). *The low-projection of a live collection register lives w.r.t. a security level σ and a live collection register labelling Ξ is given by:*

$$lives \upharpoonright^{\Xi,\sigma}_{\lightning} = \{(r, r', m) \mid \ \Xi.lives(r) \sqsubseteq \sigma\}$$

Two live collection registers $lives_0$ and $lives_1$, respectively labeled by Ξ_0 and Ξ_1, are said to be low-equal at security level σ, written $lives_0, \Xi_0 \sim^{\sigma}_{\lightning} lives_1, \Xi_1$, if they coincide in their respective low-projections, meaning that $lives_0 \upharpoonright^{\Xi_0,\sigma}_{\lightning} = lives_1 \upharpoonright^{\Xi_1,\sigma}_{\lightning}$.

Given a forest labelling Ξ, we must modify the definition of low-projection for DOM forests so that an attacker at level σ can additionally see: **(1)** the live indexes of the nodes whose position levels are $\sqsubseteq \sigma$ and **(2)** the number of descendants of visible nodes with a given tag m such that $\sigma_m \sqsubseteq \sigma$. Definiton 7.3 formally presents the new low-equality for DOM forests.

Definition 7.3 (Low-Projection and Low-Equality for DOM Forests with Live Collections). *The low-projection of a DOM forest f w.r.t. a security level σ and a labelling Ξ is given by:*

$$\begin{aligned} f \upharpoonright^{\Xi,\sigma}_{\lightning} = {} & f \upharpoonright^{\Xi,\sigma} \\ & \cup \ \{(r, m, i, r') \mid \ f \vdash r \rightsquigarrow_m \overrightarrow{r} \ \wedge \ \overrightarrow{r}(i) = r' \ \wedge \ \Xi(r').\mathsf{pos} \sqsubseteq \sigma\} \\ & \cup \ \{(r, m, n) \mid \ f \vdash r \rightsquigarrow_m \overrightarrow{r} \ \wedge \ |\overrightarrow{r}| = n \ \wedge \ \sigma_m \sqcup \Xi(r).\mathsf{node} \sqsubseteq \sigma\} \end{aligned}$$

Two DOM forests ν_0 and ν_1, respectively labeled by Ξ_0 and Ξ_1, are said to be low-equal at security level σ, written $f_0, \Xi_0 \sim^{\sigma}_{\lightning} f_1, \Xi_1$, if they coincide in their respective low-projections, meaning that $f_0 \upharpoonright^{\Xi_0,\sigma}_{\lightning} = f_1 \upharpoonright^{\Xi_1,\sigma}_{\lightning}$.

Finally, we say that two DOM states ν_0 and ν_1 respectively labelled by Ξ_0 and Ξ_1 are low-equal at a given level σ if both their corresponding forests and live collection registers are low-equal: $\nu_0.f, \Xi_0.f \sim^{\sigma}_{\lightning} \nu_1.f, \Xi_1.f$ and $\nu_0.lives, \Xi_0.lives \sim^{\sigma}_{\lightning} \nu_1.lives, \Xi_1.lives$.

7.3.4 Enforcement - Strengthening the Low-Equality for DOM forests

The new version of the low-equality for forests captures the additional power of an attacker who disposes of live collections to interact with the document. Hence, a possible way to proceed is to modify the previous monitor in order for it to enforce the stronger version of the low-equality. However, doing so would lead to stricter constraints regarding the way programs can modify the document, even if **no**

$$\textsc{Live New}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \qquad \Xi' = \langle \Xi.f, \Xi.lives\,[r \mapsto \sigma]\rangle}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^r\ \mathsf{newLive}_{lab}\ \langle \Xi', \sigma\rangle}$$

$$\textsc{Live Length}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \sigma_m \qquad \mathcal{S}ec(f, \Xi.f)}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r,m)}\ \mathsf{liveLength}\ \langle \Xi, \sigma\rangle}$$

$$\textsc{Live Item}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \Xi.f(r').\mathsf{pos} \qquad \mathcal{S}ec(f, \Xi.f)}{\langle \Xi, \sigma_0 :: \sigma_1\rangle^{(r,r')}\ \mathsf{liveIndex}_{lab}\ \langle \Xi, \sigma\rangle}$$

Figure 7.5: Extension of the Monitor to Live Primitives

live collection is used to inspect its content. Therefore, instead of imposing additional constraints on operations that update the content of the DOM forest, the new version of the monitor makes use of a predicate on DOM forests that checks **whether the inspection of the document via live collections is secure**. In a nutshell, any two labeled forests verifying this predicate and related by the first low-equality are also related by the new low-equality and, therefore, can be securely inspected using live collections.

Informally, we say that a DOM forest f labelled by Ξ is *secure for live collections*, written $\mathcal{S}ec(f, \Xi)$, if and only if: **(1)** the position level of every node in f is lower than or equal to the global position level corresponding to its tag, **(2)** the position levels of the nodes with the same tag monotonically increase in document order, and **(3)** the position level of every node is higher than or equal to the position levels of all its descendants (meaning that if the position of a node is secret, the positions of all its descendants are also secret). The predicate $\mathcal{S}ec(f, \Sigma)$ is defined with the help of a predicate $\mathcal{S}ec_{f,\Sigma} \vdash^r \phi_{\lightning} \rightsquigarrow \phi'_{\lightning}$, given in Definition 7.4, that holds if the tree rooted at r is *secure for live collections.*

Definition 7.4 (Secure Forest for Live Collections). *The predicate $\mathcal{S}ec_{f,\Sigma} \vdash^r \phi_{\lightning} \rightsquigarrow \phi'_{\lightning}$ is recursively defined as follows:*

$$\textsc{Orphan Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \\ |f(r).\mathsf{children}| = 0 \\ \phi_{\lightning}(m) \sqsubseteq \Xi(r).\mathsf{pos} \sqsubseteq \sigma_m \\ \phi'_{\lightning} = \phi_{\lightning}\,[m \mapsto \Xi(r).\mathsf{pos}] \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_{\lightning} \rightsquigarrow \phi'_{\lightning}}$$

$$\textsc{Non-Orphan Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \qquad \phi_{\lightning}(m) \sqsubseteq \Xi(r).\mathsf{pos} \sqsubseteq \sigma_m \\ |f(r).\mathsf{children}| = n > 0 \qquad \phi^0_{\lightning} = \phi_{\lightning}\,[m \mapsto \Xi(r).\mathsf{pos}] \\ \forall_{0 \le i < n}\ \Xi(r).\mathsf{pos} \sqsubseteq \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \\ \forall_{0 \le i < n}\ \mathcal{S}ec_{f,\Sigma} \vdash^{f(r).\mathsf{children}(i)} \phi^i_{\lightning} \rightsquigarrow \phi^{i+1}_{\lightning} \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_{\lightning} \rightsquigarrow \phi^n_{\lightning}}$$

In the definition above, the function $\phi_{\lightning}$ maps each tag name to the position level of the last node with that tag name preceding the node pointed to by r in f in document order. The function $\phi'_{\lightning}$ maps each tag name to the position level of the last node with that tag name in the tree rooted at r (if no such node exists, $\phi'_{\lightning}$ coincides with $\phi_{\lightning}$). Formally, the predicate $\mathcal{S}ec(f, \Sigma)$ holds if and only if for all orphan nodes pointed to by a reference r there are two functions $\phi_{\lightning}$ and $\phi'_{\lightning}$ such that $\mathcal{S}ec_{f,\Sigma} \vdash^r \phi_{\lightning} \rightsquigarrow \phi'_{\lightning}$.

7.3.5 Enforcement - Monitoring Core DOM with Live Collections

Finally, Figure 7.5 presents the monitor methods for the Core DOM API with live collections.

7.4 Related Work

7.4.1 Secure Information Flow in Dynamic Tree Structures

Russo et al. [Russo 2009] have been the first to study the problem of securing information flow in DOM-like dynamic tree structures. They present a monitor for a WHILE language with primitives for manipulating DOM-like trees and prove it sound. However, references are not modelled in this language; instead, program configurations include the current working node of the program. This is, as the authors point out, the main difference with respect to JavaScript DOM operations, since in JavaScript tree nodes are treated as first-class values. In particular, in [Russo 2009] it is not possible to change the position of a node in the DOM forest without deleting and re-creating it – its position remains the same during its whole "lifetime". Consequently, the *position level* of a node coincides with its *node level*. By treating nodes as first-class values we were able to give separate treatment to position leaks, which cannot be directly expressed in the language of [Russo 2009].

7.4.2 DOM Semantics

Gardner et al. [Gardner 2008] propose a compositional and concise formal specification of the DOM called Minimal DOM. The authors show that their semantics has no redundancy and that it is sufficient to describe the structural kernel of DOM Core Level 1, meaning that the semantics of all the other unmodelled commands can be obtained from that of the modelled ones. Additionally, they apply local reasoning based on Separation Logic and prove invariant properties of simple JavaScript programs that interact with the DOM. Given that our aim is to track information flow in the DOM, we use a simplified semantics that allows us to label DOM resources in a natural way. Like Minimal DOM, Core DOM is also compositional. Furthermore, all the primitives of Minimal DOM can be easily translated to Core DOM. Hence, we expect the authors' sufficiency claim to be applicable to Core DOM.

Chapter 8

Conclusions

Contents

In just a few decades, computational systems have dramatically evolved from the local timesharing machines of the early 70's to complex world wide webs of computing devices, where programs and data roam in a decentralized fashion. Currently, information processing is thoroughly integrated into everyday objects and activities to a point where we might engage many computational devices and systems simultaneously, while not necessarily being aware that we are doing so. However, the increasingly ubiquitous nature of computing hoists an ever widening spectrum of security vulnerabilities, making computer security an increasingly important matter.

In this scenario, many attacks arise at the application level, and can thus be tackled by means of programming language design and analysis techniques, such as static analysis or program instrumentation. As it is, in recent years, a lot of work has been dedicated to the study of information flow security in computing systems [Hedin 2011, Sabelfeld 2003a], with the double aim of **(1)** preventing classified information from falling into the hands of unauthorised parties and **(2)** preventing high-integrity resources from being updated depending on data coming from untrusted parties. However, it has been frequently observed that despite the *"ongoing attention from the research community, information-flow based enforcement mechanisms have not been widely (or even narrowly!) used"* [Zdancewic 2004]. Hence, the real challenge in Information Flow Control research is *"to find applications to all the existing results or, in failing to do so, provide a reasonable explanation for such failure"* [Zdancewic 2004]. This thesis tries to abridge this gap between theory and practice by studying a broad range of IFC mechanisms for a realistic core of a widely used programming language – JavaScript – which holds a prominent spot in the internet of today.

While the dynamic features of JavaScript make it an exceedingly difficult target for static analysis [Maffeis 2009], dynamic methods for tracking information flow often impose a runtime overhead that is far from negligible [Hedin 2014]. Hence, we consider the hybrid type system presented in Chapter 5 one of the main contributions of this thesis, as it leverages the combination of runtime and static analyses in order to overcome some of the issues of these two approaches. This type system explores a novel way of combining fully static type systems for checking secure information flow, such as those presented in [Volpano 1996] and [Banerjee 2002], with program instrumentation. Therefore, we expect that the presented method for deriving more permissive hybrid mechanisms can be replicated with advantages for other static mechanisms for securing information flow in dynamic languages.

8.1 Further Work

We envision the following tracks for future work:

Bibliography

[3rd edition of ECMA 262 1999] The 3rd edition of ECMA 262. *ECMAScript Language Specification.* Rapport technique, ECMA, 1999. (Cited on pages 1, 9, 10, 12 and 16.)

[5th edition of ECMA 262 June 2011 2011] The 5th edition of ECMA 262 June 2011. *ECMAScript Language Specification.* Rapport technique, ECMA, 2011. (Cited on pages 1, 9, 16, 20 and 63.)

[Agat 2000] Johan Agat. *Transforming out Timing Leaks.* In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM. (Cited on page 20.)

[Anderson 2005] C. Anderson, P. Giannini and S. Drossopoulou. *Towards Type Inference for JavaScript.* In ECOOP, 2005. (Cited on pages 15, 46 and 51.)

[Austin 2009] Thomas H. Austin and Cormac Flanagan. *Efficient Purely-Dynamic Information Flow Analysis.* In Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM. (Cited on pages 24, 28, 40 and 51.)

[Austin 2010] Thomas H. Austin and Cormac Flanagan. *Permissive Dynamic Information Flow Analysis.* In Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM. (Cited on pages 28, 40 and 51.)

[Austin 2012] Thomas H. Austin and Cormac Flanagan. *Multiple Facets for Dynamic Information Flow.* In Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM. (Cited on pages 23 and 51.)

[Banerjee 2002] A. Banerjee and D. A. Naumann. *Secure Information Flow and Pointer Confinement in a Java-like Language.* In CSFW, 2002. (Cited on pages 3, 11, 51 and 77.)

[Barth 2009] A. Barth, C. Jackson and J. C. Mitchell. *Securing Frame Communications in Browsers.* In Commun. ACM, 2009. (Cited on page 2.)

[Barth 2011] A. Barth. *The web origin concept.* In IETF, 2011. (Cited on page 1.)

[Barthe 2011] G. Barthe, P. R. D'Argenio and T. Rezk. *Secure information flow by self-composition.* Mathematical Structures in Computer Science, vol. 21, no. 6, pages 1207–1252, 2011. (Cited on page 20.)

[Bell 1976] D. Elliott Bell and Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations.* Rapport technique, Mitre Corp. Rep. MTR-2997 Rev. 1, 1976. (Cited on page 20.)

[Biba 1977] J. K. Biba. *Integrity Considerations for Secure Computer Systems*, 1977. (Cited on page 3.)

[Bichhawat 2014] A. Bichhawat, V. Rajani, D. Garg and C. Hammer. *Information Flow Control in WebKit's JavaScript Bytecode.* In POST, 2014. (Cited on page 40.)

[Birgisson 2012] Arnar Birgisson, Daniel Hedin and Andrei Sabelfeld. *Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing.* In 19th European Symposium on Research in Computer Security, Lecture Notes in Computer Science, pages 55–72. Springer, 2012. (Cited on page 35.)

[Bodin 2013] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt and Gareth Smith. *A Trusted Mechanised JavaScript Specification.* In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13. ACM, 2013. (Cited on page 16.)

[Charguéraud 2013] Arthur Charguéraud. *Pretty-Big-Step Semantics.* In Programming Languages and Systems, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2013. (Cited on page 16.)

[Chudnov 2010] A. Chudnov and D. A. Naumann. *Information Flow Monitor Inlining.* In CSF, 2010. (Cited on pages 23, 36 and 41.)

[Clements 2008] John Clements, Ayswarya Sundaram and David Herman. *Implementing continuation marks in JavaScript.* In Proceeding of the 9th Scheme and Functional Programming Workshop, 2008. (Cited on page 15.)

[Cohen 1977] Ellis Cohen. *Information Transmission in Computational Systems.* In Proceedings of the Sixth ACM Symposium on Operating Systems Principles, SOSP '77, pages 133–139, New York, NY, USA, 1977. ACM. (Cited on page 20.)

[Cousot 1977] P. Cousot and R. Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In POPL, pages 238–252, 1977. (Cited on page 3.)

[Crockford] D. Crockford. *ADSafe.* http://www.adsafe.org. (Cited on page 51.)

[Crockford 2008] Douglas Crockford. Javascript: The good parts. O'Reilly, 2008. (Cited on pages 9 and 14.)

[Davey 2002] B. A. Davey and H. A. Priestley. Introduction to lattices and order (2. ed.). Cambridge University Press, 2002. (Cited on page 3.)

[Denning 1976] Dorothy E. Denning. *A Lattice Model of Secure Information Flow.* Commun. ACM, vol. 19, no. 5, pages 236–243, May 1976. (Cited on page 20.)

[Devriese 2010] Dominique Devriese and Frank Piessens. *Noninterference through Secure Multi-execution.* In Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 23.)

[Disney 2011] T. Disney and C. Flanagan. *Gradual Information Flow Typing.* In STOP, 2011. (Cited on page 51.)

[FBJS] The FaceBook Team: FBJS. http://wiki.developers.facebook.com/index.php/FBJS. (Cited on page 51.)

[Fennell 2013] L. Fennell and P. Thiemann. *Gradual Security Typing with References.* In CSF, 2013. (Cited on page 51.)

[Flanagan 2011] David Flanagan. Javascript - the definitive guide. O'Reilly, 2011. (Cited on page 9.)

[Gardner 2008] P. Gardner, G. Smith, M. J. Wheelhouse and U. Zarfaty. *DOM: Towards a Formal Specification.* In PLAN-X, 2008. (Cited on page 75.)

[Goguen 1982] Joseph A. Goguen and José Meseguer. *Security Policies and Security Models.* In IEEE Symposium on Security and Privacy, pages 11–20, 1982. (Cited on pages 3, 20 and 51.)

[Grosskurth 2005] Alan Grosskurth and Michael W. Godfrey. *A Reference Architecture for Web Browsers.* In Proceedings of the 21st International Conference on Software Maintenance, ICSM '05, pages 661–664, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 4.)

[Guernic 2007] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007. (Cited on page 51.)

[Guha 2010] A. Guha, C. Saftoiu and S. Krishnamurthi. *The Essence of Javascript.* In ECOOP, 2010. (Cited on page 16.)

[Guha 2012] A. Guha, B. Lerner, J. Gibbs Politz and S. Krishnamurthi. *Web API Verification: Results and Challenges.* 2012. (Cited on pages 4 and 53.)

[Hedin 2011] D. Hedin and A. Sabelfeld. *A Perspective on Information Flow Control.* Marktoberdorf, 2011. (Cited on pages 3 and 77.)

[Hedin 2012] Daniel Hedin and Andrei Sabelfeld. *Information-Flow Security for a Core of JavaScript.* In Proceedings of the 25th IEEE Computer Security Foundations Symposium, CSF'12, pages 3–18. IEEE, 2012. (Cited on pages 18, 20, 21, 23, 24, 35 and 40.)

[Hedin 2014] D. Hedin, B. Birgisson, L. Bello and A. Sabelfeld. *JSFlow: Tracking Information Flow in JavaScript and its APIs.* In SAC, 2014. (Cited on pages 4 and 77.)

[Jang 2009] Dongseok Jang and Kwang-Moo Choe. *Points-to Analysis for JavaScript.* In Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pages 1930–1937. ACM, 2009. (Cited on page 15.)

[Jang 2010] D. Jang, R. Jhala, S. Lerner and H. Shacham. *An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications.* In CCS, 2010. (Cited on page 1.)

[Jensen 2009] Simon Holm Jensen, Anders Møller and Peter Thiemann. *Type Analysis for JavaScript.* In Proceedings of the 16th International Static Analysis Symposium (SAS), volume 5673 of *LNCS*, pages 238–255. Springer-Verlag, August 2009. (Cited on page 15.)

[Li 2003] P. Li, Y. Mao and S. Zdancewic. *Information Integrity Policies.* In Formal Aspects in Security & Trust (FAST), 2003. (Cited on page 3.)

[Louw 2012] M. T. Louw, K. T. Ganesh and V. N. Venkatakrishnan. *AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements.* In USENIX Security'10, 2012. (Cited on page 2.)

[Luo 2012] Z. Luo and T. Rezk. *Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication.* In CSF, 2012. (Cited on pages 2 and 15.)

[Maffeis 2008] Sergio Maffeis, John C. Mitchell and Ankur Taly. *An Operational Semantics for JavaScript.* In Proceedings of the Asian Symposium on Programming Languages and Systems, volume 5356 of *LNCS*, pages 307–325, 2008. (Cited on pages 12 and 16.)

[Maffeis 2009] S. Maffeis and A. Taly. *Language-Based Isolation of Untrusted JavaScript.* In CSF, 2009. (Cited on pages 4, 43, 51 and 77.)

[Magazinius 2010] J. Magazinius, A. Askarov and A. Sabelfeld. *A Lattice-based Approach to Mashup Security.* In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10, pages 15–23, New York, NY, USA, 2010. ACM. (Cited on page 2.)

[Magazinius 2012] J. Magazinius, A. Russo and A. Sabelfeld. *On-the-fly Inlining of Dynamic Security Monitors.* Computers & Security, 2012. (Cited on pages 23, 36 and 41.)

[Matos 2005] A. Almeida Matos and G. Boudol. *On Declassification and the Non-Disclosure Policy.* In CSFW, 2005. (Cited on pages 3 and 18.)

[Moore 2011] S. Moore and S. Chong. *Static Analysis for Efficient Hybrid Information-Flow Control.* In CSF, 2011. (Cited on page 51.)

[Politz 2011] J. Gibbs Politz, S. A. Eliopoulos, A. Guha and S. Krishnamurthi. *ADsafety: Type-Based Verification of JavaScript Sandboxing.* In USENIX Security Symposium, 2011. (Cited on page 51.)

[Pottier 2003] F. Pottier, and V. Simonet. *Information Flow Inference for ML.* ACM Trans. Program. Lang. Syst., 2003. (Cited on pages 3 and 51.)

[Recommendation 2000] W3C Recommendation. *DOM: Document Object Model (DOM) Level 1 Specification (2nd Ed.).* Rapport technique, W3C, 2000. (Cited on page 4.)

[Recommendation 2005] W3C Recommendation. *DOM: Document Object Model (DOM).* Rapport technique, W3C, 2005. (Cited on pages 4, 63 and 69.)

[Richards 2010] Gregor Richards, Sylvain Lebresne, Brian Burg and Jan Vitek. *An Analysis of the Dynamic Behavior of JavaScript Programs.* vol. 45, pages 1–12, 2010. (Cited on page 51.)

[Russo 2009] A. Russo, A. Sabelfeld and A. Chudnov. *Tracking Information Flow in Dynamic Tree Structures.* In ESORICS, 2009. (Cited on pages 5, 6, 63, 74 and 75.)

[Russo 2010] A. Russo and A. Sabelfeld. *Dynamic vs. Static Flow-Sensitive Security Analysis.* In CSF, 2010. (Cited on pages 40, 41, 51 and 56.)

[Sabelfeld 2003a] A. Sabelfeld and A. C. Myers. *Language-Based Information-Flow Security.* IEEE Journal on Selected Areas in Communications, 2003. (Cited on pages 3, 4, 24, 45, 46 and 77.)

[Sabelfeld 2003b] A. Sabelfeld and A. C. Myers. *A Model for Delimited Information Release.* In ISSS, 2003. (Cited on page 2.)

[Sabelfeld 2009] A. Sabelfeld and A. Russo. *From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research.* In Ershov Memorial Conference, 2009. (Cited on page 56.)

[Shroff 2007] P. Shroff, S. F. Smith and M. Thober. *Dynamic Dependency Monitoring to Secure Information Flow.* In CSF, 2007. (Cited on page 51.)

[Taly 2011] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller and J. Nagra. *Automated Analysis of Security-Critical JavaScript APIs.* In SP, 2011. (Cited on pages 44 and 60.)

[Thiemann 2005] P. Thiemann. *Towards a Type System for Analyzing JavaScript Programs.* In ESOP, 2005. (Cited on pages 15, 16 and 51.)

[Venkatakrishnan 2006] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney and R. Sekar. *Provably Correct Runtime Enforcement of Non-interference Properties.* In ICICS, 2006. (Cited on page 51.)

[Volpano 1996] Dennis M. Volpano, Cynthia E. Irvine and Geoffrey Smith. *A Sound Type System for Secure Flow Analysis.* Journal of Computer Security, vol. 4, no. 2-3, pages 167–187, January 1996. (Cited on pages 3, 20, 51 and 77.)

[Yang 2013] E. Yang, D. Stefan, J. Mitchell, D. Mazières, P. Marchenko and B. Karp. *Toward Principled Browser Security.* In 14th Workshop on Hot Topics in Operating Systems, Berkeley, CA, 2013. USENIX. (Cited on pages 2 and 3.)

[Zdancewic 2002] Stephan Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, Ithaca, New York, August 2002. (Cited on pages 24 and 28.)

[Zdancewic 2004] S. Zdancewic. *Challenges for information-flow security.* In Programming Language Interference and Dependence (PLID), 2004. (Cited on page 77.)

Appendix A

Name of Appendix A