Hybrid Typing of Secure Information Flow in a JavaScript-like Language

José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt

Abstract

As JavaScript is highly dynamic by nature, static information flow analyses are often too coarse to deal with the dynamic constructs of the language. To cope with this challenge, we present and prove the soundness of a new hybrid typing analysis for securing information flow in a JavaScript-like language. Our analysis combines static and dynamic typing in order to avoid rejecting programs due to imprecise typing information. Program regions that cannot be precisely typed at static time are wrapped inside an internal *boundary* statement used by the semantics to interleave the execution of statically verified code with the execution of code that must be dynamically checked.

1 Introduction

The dynamic aspects of JavaScript make the analysis of JavaScript programs very challenging. On one hand, one may use a purely static analysis, but either restrict the language to exclude these dynamic aspects or over-approximate them; this is too coarse to be applicable in practice. On the other hand, one may use purely dynamic mechanisms, such as monitoring or secure multi-executions [1, 7, 9, 17]; but the gained precision comes at the cost of a much lower performance compared to the original code [8].

The work on staged information flow [2] combines static and dynamic analyses to handle `eval` operations more precisely, but it does not deal with other dynamic features of the language. In contrast, we propose a general hybrid analysis to statically verify secure information flow in a core of JavaScript. Following the hybrid typing motto "static analysis where possible with dynamic checks where necessary" [6], we are able to reduce the runtime overhead introduced by purely dynamic analyses without excluding dynamic field operations. In fact, our analysis can handle some of the most challenging JavaScript features, such as prototype-based inheritance, extensible objects, and constructs for checking the existence of object properties. Its key ingredient is an internal boundary statement inspired by recent work in inter-language interoperability [11]. The static component of our analysis wraps program regions that cannot be precisely verified inside an internal *boundary* statement instead of rejecting the whole program. This boundary statement identifies the regions of the program that must be verified at runtime—which may be as small as a single statement—and enables the initial set up required by the dynamic analysis. In summary, the proposed boundary statement allows the semantics to effortlessly interleave the execution of statically verified code with the execution of code that must be verified at runtime.

Although our work is generally motivated by the verification of dynamic features of JavaScript, we choose to focus on the particular case of constructs that rely on dynamic computation of object field names, which we call *dynamic field operations*. In JavaScript, one can access a field `f` of an object `o` either by writing `o.f` or `o[e]`, where `e` is an expression that dynamically evaluates to the string `f`. Dynamic computation of field names is one of the major sources of imprecision of static analyses for JavaScript [10].

This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008 and the EPSRC Grant Reference EP/H008373/1.

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$v \in \mathcal{V}al$	::=	$lit \mid \underline{lit} \mid l \mid \lambda x : \dot{\tau}.s$
$e \in \mathcal{E}xpr$	::=	$v \mid \mathsf{this} \mid x \mid x = e \mid \{\,\}\,[\dot{\tau}] \mid e.f \mid e_1[e_2] \mid e_1.f = e_2$
		$\mid e_1[e_2] = e_3 \mid f \,\mathsf{in}\, e \mid [e_1] \,\mathsf{in}\, e_2 \mid \mathsf{delete}\, e.f \mid \mathsf{delete}\, e_1[e_2]$
		$\mid \mathsf{function}\,(x)[\dot{\tau}]\{s\} \mid e_1(e_2) \mid e_1.x(e) \mid e_1[e_2](e_3)$
$s \in \mathcal{S}tmt$	::=	$e \mid \mathsf{var}\, x\,[\dot{\tau}] \mid s_1;\, s_2 \mid \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \mid \mathsf{return}\, e$

Table 1 Core JS Syntax - Expressions and Statements

▶ **Example 1** (Running example: the challenge of typing dynamic field operations)**.** Listing 1 presents a program that creates an object `o` with a secret field `secret1` and two public fields `public1` and `public2`.

```
o = {}; o.secret1 = secret_input();
o.public1 = public_input(); o.public2 = public_input(); public = o[g()]
```

Listing 1 Running Example - Potentially Insecure Program

The secret field `secret1` gets a secret input via function `secret_input`, while the two public fields `public1` and `public2` each get a public input via function `public_input`. The program then assigns the value of one of the three fields to the public variable `public`, as determined by the return value of function `g`. Concretely, when `g` returns the string `"secret1"`, the program assigns a secret value to `public` and the execution is insecure. On the other hand, when `g` returns either `"public1"` or `"public2"`, the program assigns a public value to `public` and the execution is secure. However, in order to make sure that `g` never returns `"secret1"`, a static analysis needs to predict the dynamic behaviour of `g`, which is, in general, undecidable.

The loss of precision introduced by the dynamic computation of field names is not exclusive to field projections. It also occurs in method calls, field deletions, and membership checks. We account for the use of these operations by verifying them at runtime. When verifying a statement containing a dynamic field operation, the static component of the analysis wraps it inside a boundary statement. In the case of the running example, all statements except the last one are statically typed. In contrast, the last assignment is re-written as `@monitor(@type_env, @pc, @ret, public = o[g()])`, where the first three arguments of the monitor statement are used for the setup of the runtime analysis. Hence, when the program is executed the only overhead introduced by the dynamic component of our hybrid analysis regards the security checks for validating or rejecting the statement `public = o[g()]`.

Contributions. The main contribution of the paper is the design of a new hybrid analysis for verifying secure information flow in a JavaScript-like language. To achieve this, we introduce: **(1)** a type language specifically designed to control information flow in a subset of JavaScript, **(2)** a static type system for verifying statements not containing dynamic field operations, **(3)** a dynamic typing analysis for verifying statements containing dynamic field operations, and **(4)** a novel boundary operator for interleaving the execution of statically verified regions with dynamically verified ones. Finally, we have implemented a prototype as well as a case study, available online at [16].

2 Core JS

Syntax. The syntax of Core JS is given in Table 1. Expressions include values, the keyword this, variables, variable assignments, object literals, static and dynamic field projections, static and dynamic field assignments, static and dynamic membership checks, static and dynamic field deletions, function literals, function calls, and static and dynamic method calls. Statements include expression statements, variable declarations, sequences, condi-

$\hat{E}$	::=	$\square \mid x = \hat{E} \mid \hat{E}.f \mid \hat{E}[e] \mid l[\hat{E}] \mid \hat{E}.f = e \mid \hat{E}[e_1] = e_2$
		$\mid l[\hat{E}] = e \mid l[f] = \hat{E} \mid [\hat{E}]\ \mathsf{in}\ e \mid [f]\ \mathsf{in}\ \hat{E} \mid \mathsf{delete}\ \hat{E}.f \mid \mathsf{delete}\ \hat{E}[e]$
		$\mid \mathsf{delete}\ l[\hat{E}] \mid \hat{E}(e) \mid l(\hat{E}) \mid \hat{E}.f(e) \mid \hat{E}[e](e) \mid l[\hat{E}](e) \mid l[f](\hat{E})$
E	::=	$\hat{E} \mid E;\ s \mid \mathsf{if}(\hat{E})\ \{s_1\}\ \mathsf{else}\ \{s_2\} \mid \mathsf{return}\ \hat{E}$

Table 2 Evaluation Contexts

tional statements, and return statements. We distinguish two types of *values*: literal values and runtime values. Literal values include numbers, booleans, strings, and undefined. Runtime values, ranged over by $\underline{v}$, include parsed literal values, locations, and parsed function literals. Object literals, function literals, and variable declarations are annotated with their respective *security types* (which are explained in Section 3). In the following, we denote by $\mathcal{E}xpr_{\lightning}$ the set of Core JS expressions that contain dynamic field operations.

Memory Model. A heap $H \in \mathcal{H}eap : \mathcal{L}oc \times \mathcal{X} \rightharpoonup \mathcal{V}al$ is a partial mapping from locations in $\mathcal{L}oc$ and field names in $\mathcal{X}$ to values in $\mathcal{V}al$. We denote a heap cell by $(l, f) \mapsto v$, the union of two disjoint heaps by $H_1 \uplus H_2$, a read operation by $H(l, f)$, and a heap update operation by $H[l.f \mapsto v]$. An object can be seen as a set of heaps cells addressed by the same location but with different field names. We use $l \mapsto \{f_1 : v_1, \ldots, f_n : v_n\}$ as an abbreviation for the object $(l, f_1) \mapsto v_1 \uplus \ldots \uplus (l, f_n) \mapsto v_n$.

Every object has a prototype, whose location is stored in a special field *__proto__*. In order to determine the value of a field f of an object o, the semantics first checks whether f is one of the fields of o. If that is the case, the field look-up yields that value. Otherwise, the semantics checks whether f belongs to the fields of the prototype of o 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. The prototype chain inspection procedure is modelled by the semantic function π given in appendix. Informally, the expression $\pi(H, l, f)$ denotes the location of the first object that defines f in the prototype chain of the object pointed to by l. Given that most implementations of JavaScript allow for explicit prototype mutation, we include this feature in Core JS. For instance, $x.\_\_proto\_\_$ evaluates to the the prototype of the object bound to x and $x.\_\_proto\_\_ = y$ sets the prototype of the object bound to x to the object bound to y.

Scope is modelled using *environment records*. An environment record is simply an internal object that maps variable names to their respective values. An environment record is created for every function or method call. We use $\mathsf{act}(l, x, v, s, l')$ to denote the environment record that: **(1)** is identified by location l where it is stored, **(2)** maps x to v, **(3)** maps all the variables declared in s to undefined, and **(4)** maps the field @*this* to the location l'. (Note that environment records map a single variable because functions have a single argument. Moreover, in the execution of a method call, the field @*this* is used to store the location of the object on which the method was invoked.) Variables are resolved with respect to a list of environment record locations, called *scope chain*. The variable inspection procedure is modelled by the semantic function σ given in appendix. We let $\sigma(H, L, x)$ denote the location of the first environment record that defines x in the scope chain L. The global object, assumed to be pointed to by a fixed location l_g, is the environment record that binds global variables. Since functions are first-class citizens, the evaluation of a function literal triggers the creation of a special type of object, called *function object*. Every function object has two fields: @*body* and @*scope*, which respectively store the corresponding parsed function literal and the scope chain that was active when the function literal was evaluated.

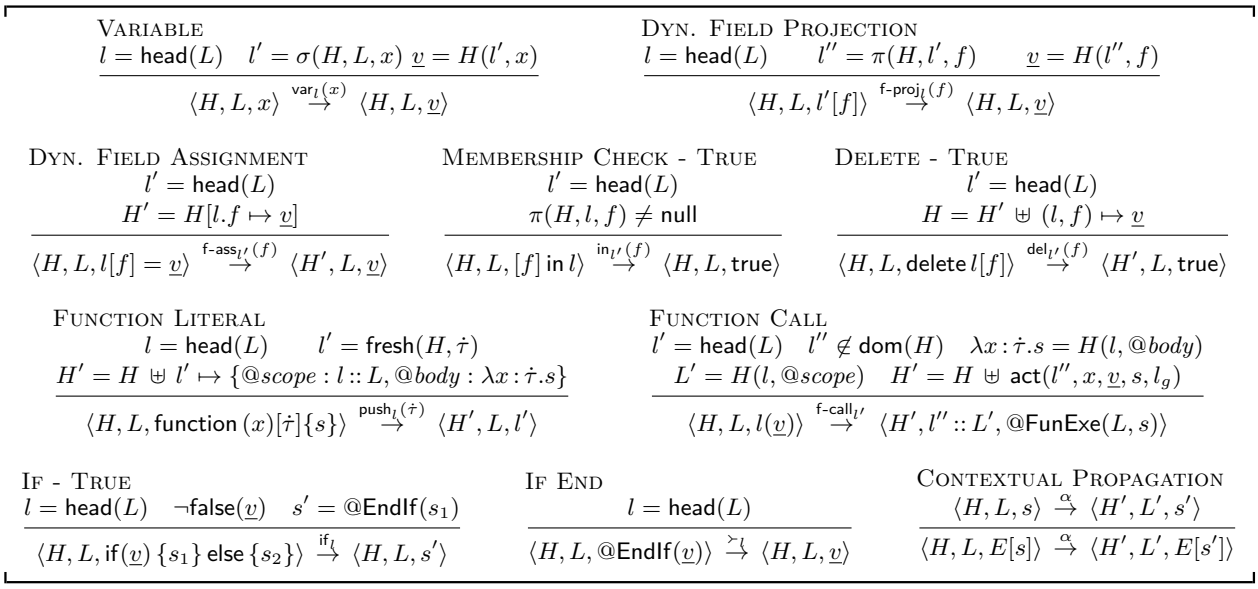

Figure 1 Fragment of the Small-Step Semantics of Core JS

Functions execute in the scope in which the they were evaluated.

Semantics. Figure 1 presents a fragment of the semantics in the style of Wright and Felleisen [20] (the full semantics is given in appendix). A configuration Ψ has the form $\langle H, L, s\rangle$ where H is the current heap, L the current scope chain, and s the statement to execute. Transitions are labelled with an internal event α for the use of the dynamic analysis. The evaluation order is specified with the help of evaluation contexts, whose syntax is given in Table 2. In the following, we use $l :: L$ for the list obtained by prepending l to L and $\mathsf{head}(L)$ for the first element of L.

Rule Variable uses σ to determine the location l' of the environment record that defines x and reads its value from the heap. Rule Dyn Field Projection uses π to determine the location l'' of the object that defines f in the prototype chain of the object pointed to by l' and then reads that field's value from the heap. Rule Dyn Field Assignment updates the current heap with a mapping from l and f to $\underline{v}$. Rule Membership Check - True checks if f is defined in the prototype chain of the object pointed to by l and evaluates to true. Rule Delete - True removes the cell $(l, f) \mapsto v$ from the heap and evaluates to true. Rule Function Literal adds a new function object to the heap. Rule Function Call extends the heap with a new environment record for the evaluation of the function pointed to by l. The current scope chain L is replaced with the scope chain L' that was active when the corresponding function literal was evaluated extended with the location l'' of the newly created environment record. The semantics makes use of an internal statement $@\mathsf{FunExe}(L, s)$ for keeping track of the caller's scope chain during the execution of the function's body. Rule If - True checks if the guard of the conditional does not belong to the set of *falsy* values –$\{\mathsf{false}, 0, \mathsf{undefined}, \mathsf{null}\}$– and replaces the whole conditional with its then-branch followed by an internal statement $@\mathsf{EndIf}$ used for notifying the dynamic analysis of the end of that branch.

3 Static Typing Secure Information Flow in Core JS

In this section, we present both a new type language for controlling information flow in JavaScript and the static component of our analysis. Here, the specification of security

policies relies on two key elements: a lattice of security levels and a typing environment that maps resources to security types, which can be viewed as safety types annotated with security levels. In the examples, we use $\mathcal{L} = \{H, L\}$ with $L \sqsubset H$, meaning that L-labelled resources (*low* resources) are less confidential than those labelled with H (*high*). We use $\sqcup$, $\bot$, and $\top$ for the least upper bound (*lub*), the *bottom* level, and the *top* level, respectively.

Security Types. A security type $\dot{\tau}$ is obtained by pairing up a *raw type* τ with a security level σ, called its *external level*. The external level of a security type establishes 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 $\textsf{PRIM}^L$ may only depend on *low* resources. The syntax of raw types is given and explained below:

$$\tau ::= \textsf{PRIM} \mid \ \langle \dot{\tau}.\dot{\tau} \xrightarrow{\sigma} \dot{\tau} \rangle \mid \ \langle \kappa.\dot{\tau} \xrightarrow{\sigma} \dot{\tau} \rangle \mid \mu\kappa.\langle f^{\sigma} : \dot{\tau}, \cdots, f^{\sigma} : \dot{\tau}, *^{\sigma} : \dot{\tau} \rangle \mid \mu\kappa.\langle f^{\sigma} : \dot{\tau}, \cdots, f^{\sigma} : \dot{\tau} \rangle$$

- The type $\textsf{PRIM}$ is the type of expressions which evaluate to primitive values.
- The type $\langle \dot{\tau}_0.\dot{\tau}_1 \xrightarrow{\sigma} \dot{\tau}_2 \rangle$ is the type of expressions which evaluate to 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 $\dot{\tau}_0$. Level σ is the *writing effect* [15] of functions of this type, that is, a lower bound on the levels of the resources updated or created during their execution. When specifying a function type inside an object type, one can use the type variable bound by that object type as the type of the keyword this (in the syntax of types, κ ranges over the set of type variables).
- The type $\mu\kappa.\langle f_0^{\sigma_0} : \dot{\tau}_0, \cdots, f_n^{\sigma_n} : \dot{\tau}_n, *^{\sigma_*} : \dot{\tau}_* \rangle$ is the type of expressions which evaluate to objects that **may** define the fields f_0 to f_n mapping each field f_i to a value of security type $\dot{\tau}_i$. The security type assigned to $*$ is the *default security type*, which is the security type of all fields not in $\{f_0, \cdots, f_n\}$. Every field f_i is further associated with an *existence level* σ_i that establishes an upper bound on the levels of the contexts in which the field can be created or deleted. The level σ_* is the *default existence level*. When no default security type is declared, the objects of the type may only define explicitly declared fields. The reason why we do not precisely track the presence of fields in object types is that we do not want the type of an object to change at runtime even though its structure may change. Furthermore, the absence of a field in a type does not mean it cannot be accessed in objects of that type: this field may still be defined in the prototype chain. We could have flattened security types for objects by requiring every object type to explicitly declare all the fields accessible through the prototype chains of the objects of that type, but this would have two disadvantages. First, object types would be less precise, and second, they would be much larger as the types of prototype fields would be duplicated. The cost of this design choice is a more complex Static Field Projection typing rule that has to take the prototype chain into account.

Given a security type $\dot{\tau}$, the expression $\mathsf{lev}(\dot{\tau})$ denotes its external level and $\lfloor \dot{\tau} \rfloor$ its raw type (for instance, $\mathsf{lev}(\textsf{PRIM}^L) = L$ and $\lfloor \textsf{PRIM}^L \rfloor = \textsf{PRIM}$). We define $\dot{\tau}^{\sigma}$ as $\lfloor \dot{\tau} \rfloor^{\mathsf{lev}(\dot{\tau}) \sqcup \sigma}$ (for example, $(\textsf{PRIM}^L)^H = \textsf{PRIM}^H$). Given a function security type $\dot{\tau} = \langle \dot{\tau}_0.\dot{\tau}_1 \xrightarrow{\sigma} \dot{\tau}_2 \rangle^{\sigma'}$, we use $\dot{\tau}.\mathsf{this}$, $\dot{\tau}.\mathsf{arg}$, $\dot{\tau}.\mathsf{ret}$, and $\dot{\tau}.\mathsf{wef}$ to denote $\dot{\tau}_0$, $\dot{\tau}_1$, $\dot{\tau}_2$, and σ, respectively. Given an object security type $\dot{\tau}$, we use $\mathsf{dom}(\dot{\tau})$ for the set containing all field names explicitly declared in $\dot{\tau}$ (including $*$, if present). Given a field name f and an object security type $\dot{\tau}$, $\dot{\tau}.f$ ($\dot{\tau}.\overline{f}$, resp.) denotes either the security type (existence level resp.) with which $\dot{\tau}$ associates f or its default security type (existence level, resp.) when $f \notin \mathsf{dom}(\dot{\tau})$ and $* \in \mathsf{dom}(\dot{\tau})$. The ordering $\sqsubseteq$ on security levels induces a simple ordering $\preceq$ on security types: $\dot{\tau}_0 \preceq \dot{\tau}_1$ *iff* $\mathsf{lev}(\dot{\tau}_0) \sqsubseteq \mathsf{lev}(\dot{\tau}_1)$ and $\lfloor \dot{\tau}_0 \rfloor = \lfloor \dot{\tau}_1 \rfloor$. We use $\dot{\tau}_g$ for the type of the global object. Finally, a typing environment Γ is simply a mapping from variables to security types.

$$
\begin{array}{rcl}
\Gamma(\texttt{public}) & = & \textsc{prim}^{L} \\
\Gamma(\texttt{secret}) & = & \textsc{prim}^{H} \\
\Gamma(\texttt{secret\_input}) & = & \langle \dot{\tau}_g.\_ \xrightarrow{H} \textsc{prim}^{H} \rangle^{L} \\
\Gamma(\texttt{public\_input}) & = & \langle \dot{\tau}_g.\_ \xrightarrow{H} \textsc{prim}^{L} \rangle^{L} \\
\Gamma(\texttt{g}) & = & \langle \dot{\tau}_g.\_ \xrightarrow{H} \textsc{prim}^{L} \rangle^{L}
\end{array}
\qquad
\dot{\tau}_o = \mu\kappa \cdot \left\langle \begin{array}{l} \texttt{public1}^{L} : \textsc{prim}^{L}, \\ \texttt{public2}^{L} : \textsc{prim}^{L}, \\ \texttt{secret1}^{H} : \textsc{prim}^{H} \\ \texttt{secret2}^{H} : \textsc{prim}^{H} \end{array} \right\rangle^{L}
$$

$$
\Gamma(\texttt{o0}) = \mu\kappa.\langle \texttt{\_proto\_}^{H} : \dot{\tau}_o \rangle^{L}
$$

$$
\Gamma(\texttt{o}) = \Gamma(\texttt{o1}) = \Gamma(\texttt{o2}) = \dot{\tau}_o
$$

Table 3 Typing Environment for the Examples of Listings 1, 2, and 3 .

▶ **Example 2.** Table 3 presents the typing environment used to type the programs given in Listings 1, 2, and 3. Since `secret_input`, `public_input`, and `g` are to be used as functions, their respective types use the type of the global object as the type of the keyword this. Since none of these three functions expects an argument or updates the heap, their respective types omit the type of the argument and declare a *high* writing effect. Our design choice of not flattening object types can also be seen in this example: the type of `o0` is much shorter as it does not need to mention at top level the fields declared in $\dot{\tau}_o$.

Static Type System. The key insight of the static type system is that it wraps program regions which cannot be precisely analysed at static time within a boundary statement $@monitor(\Gamma, pc, \dot{\tau}_r, s)$ responsible for turning on the typing analysis at runtime. The parameters Γ, pc, and $\dot{\tau}_r$ are the typing environment, the context level [15], and the type of the function whose body is being typed, respectively. Given a typing environment Γ, a level pc, and an expression e, the typing judgment $\Gamma, pc \vdash_e e \hookrightarrow e' : \dot{\tau}$ means that e is rewritten as a semantically equivalent expression e', which may include boundary statements, has raw type $\lfloor \dot{\tau} \rfloor$, and reads variables or fields of level at most $\mathsf{lev}(\dot{\tau})$. Typing judgements for statements, with the form $\Gamma, pc, \dot{\tau}_r \vdash_s s \hookrightarrow s'$, differ from typing judgements for expressions in that they do not assign a type to the statement. When e (s resp.) coincides with e' (s' resp.), we omit $\hookrightarrow e'$ ($\hookrightarrow s'$ resp.) from the typing rules. The typing rules are given in Figure 2 and described below.

Static field projection As a given field may be defined anywhere in the prototype chain of the inspected object, this rule needs to take into account the whole prototype chain of that object. To this end, we overload function π to model a static prototype chain inspection procedure. Informally, $\pi(\dot{\tau}, f)$ computes the *lub* between the security types of f in the prototype chain of objects of type $\dot{\tau}$ and upgrades the external level of this type with the *lub* between the existence levels of the field *__proto__* in that prototype chain.

▶ **Example 3** (Leaks via Prototype Mutations)**.** The program below creates three empty objects: `o0`, `o1`, and `o2`. Then, it creates a field named `public1` in both `o1` and `o2`, which is set to 0 in `o1` and to 1 in `o2`. Depending on the value of a *high* variable `secret`, the prototype of `o0` is either set to `o1` or to `o2`. Finally, the *low* variable `public1` is set to the value of the field `public1` of the prototype of `o0` (because `o0` does not define that field), thereby creating an implicit information flow between `secret` and `public`.

```
o0 = {}; o1 = {}; o2 = {}; o1.public1 = 0; o2.public1 = 1;
if (secret) { o0._proto_ = o1 } else { o0._proto_ = o2 }; public = o.public1
```

Listing 2 Security Leak via Prototype Mutation

Letting Γ be the typing environment of Table 3, it follows that $\pi(\Gamma(\texttt{o0}), \texttt{public1}) = \textsc{prim}^{H}$ because $\Gamma(\texttt{o0}).\overline{\texttt{\_proto\_}} = H$. Hence, the assignment `public = o0.public1` is not typable as the type of `o0.public1`, $\textsc{prim}^{H}$, is not lower than or equal to $\textsc{prim}^{L}$.

$$\textsc{Literal}\ \frac{}{\Gamma, pc \vdash_e lit : \mathsf{PRIM}^{\bot}} \qquad \textsc{This}\ \frac{}{\Gamma, pc \vdash_e \mathsf{this} : \Gamma(this)} \qquad \textsc{Variable}\ \frac{}{\Gamma, pc \vdash_e x : \Gamma(x)}$$

$$\textsc{Assignment}\ \frac{\Gamma, pc \vdash_e e : \dot\tau \qquad \dot\tau^{pc} \preceq \Gamma(x)}{\Gamma, pc \vdash_e x = e : \dot\tau} \qquad \textsc{Object Literal}\ \frac{pc \sqsubseteq \mathsf{lev}(\dot\tau)}{\Gamma, pc \vdash_e \{\,\}\,[\dot\tau] : \dot\tau} \qquad \textsc{Static Field Projection}\ \frac{\Gamma, pc \vdash_e e : \dot\tau \qquad \dot\tau_f = \pi(\dot\tau, f)}{\Gamma, pc \vdash_e e.f : \dot\tau_f^{\mathsf{lev}(\dot\tau)}}$$

$$\textsc{Static Member Check}\ \frac{\Gamma, pc \vdash_e e : \dot\tau \qquad \sigma = \mathsf{lev}(\dot\tau) \sqcup \bar\pi(\dot\tau, f)}{\Gamma, pc \vdash_e f\ \mathsf{in}\ e : \mathsf{PRIM}^{\sigma}} \qquad \textsc{Static Field Assignment}\ \frac{\forall_{i=1,2}\, \Gamma, pc \vdash_e e_i : \dot\tau_i \qquad \dot\tau_2 \preceq \dot\tau_1.f \qquad pc \sqcup \mathsf{lev}(\dot\tau_1) \sqsubseteq \dot\tau_1.\overline{f}}{\Gamma, pc \vdash_e e_1.f = e_2 : \dot\tau_2}$$

$$\textsc{Static Field Deletion}\ \frac{\Gamma, pc \vdash_e \mathsf{delete}\, e : \dot\tau \qquad pc \sqcup \mathsf{lev}(\dot\tau) \sqsubseteq \dot\tau.\overline{f} = \sigma_f}{\Gamma, pc \vdash_e \mathsf{delete}\, e.f : \mathsf{PRIM}^{\sigma_f}}$$

$$\textsc{Function Literal}\ \frac{\Gamma' = \mathsf{hoist}(\Gamma[x \mapsto \dot\tau.\mathsf{arg}, this \mapsto \dot\tau.\mathsf{this}], s) \qquad pc' = \dot\tau.\mathsf{wef} \qquad \mathsf{lev}(\dot\tau) \sqcup pc \sqsubseteq pc' \qquad \Gamma', pc', \dot\tau \vdash_s s \hookrightarrow s'}{\Gamma, pc \vdash_e \mathsf{function}\,(x)[\dot\tau]\{s\} \hookrightarrow \mathsf{function}\,(x)[\dot\tau]\{s'\} : \dot\tau}$$

$$\textsc{Function Call}\ \frac{\forall_{i=1,2}\, \Gamma, pc \vdash_e e_i : \dot\tau_i \qquad \sigma = pc \sqcup \mathsf{lev}(\dot\tau_1) \qquad \sigma \sqsubseteq \dot\tau_1.\mathsf{wef} \qquad \dot\tau_g^{\sigma} \preceq \dot\tau_1.\mathsf{this} \qquad \dot\tau_2^{\sigma} \preceq \dot\tau_1.\mathsf{arg}}{\Gamma, pc \vdash_e e_1(e_2) : (\dot\tau_1.\mathsf{ret})^{\sigma}}$$

$$\textsc{Static Method Call}\ \frac{\forall_{i=1,2}\, \Gamma, pc \vdash_e e_i : \dot\tau_i \qquad \dot\tau_f = \pi(\dot\tau_1, f) \qquad \sigma = pc \sqcup \mathsf{lev}(\dot\tau_1) \sqcup \mathsf{lev}(\dot\tau_f) \qquad \sigma \sqsubseteq \dot\tau_f.\mathsf{wef} \qquad \dot\tau_1^{\sigma} \preceq \dot\tau_f.\mathsf{this} \qquad \dot\tau_2^{\sigma} \preceq \dot\tau_f.\mathsf{arg}}{\Gamma, pc \vdash_e e_1.f(e_2) : (\dot\tau_f.\mathsf{ret})^{\sigma}}$$

$$\textsc{Verified Expr Stmt}\ \frac{\Gamma, pc \vdash_e e \hookrightarrow e' : \dot\tau}{\Gamma, pc, \dot\tau_{ret} \vdash_s e \hookrightarrow e'}$$

$$\textsc{Dyn. Expression Stmt}\ \frac{e \in \mathcal{E}xpr_{\lightning} \qquad s = @monitor(\Gamma, pc, \dot\tau_r, e)}{\Gamma, pc, \dot\tau_r \vdash_s e \hookrightarrow s}$$

$$\textsc{(Partially) Verified Conditional}\ \frac{\Gamma, pc \vdash_e e \hookrightarrow e' : \dot\tau \qquad \forall_{i=0,1}\, \Gamma, pc \sqcup \mathsf{lev}(\dot\tau), \dot\tau_r \vdash_s s_i \hookrightarrow s_i'}{\Gamma, pc, \dot\tau_{ret} \vdash_s \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \hookrightarrow \mathsf{if}(e')\,\{s_1'\}\,\mathsf{else}\,\{s_2'\}}$$

$$\textsc{Monitored Conditional}\ \frac{e \in \mathcal{E}xpr_{\lightning} \qquad s = @monitor(\Gamma, pc, \dot\tau_{ret}, \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\})}{\Gamma, pc, \dot\tau_{ret} \vdash_s \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \hookrightarrow s}$$

$$\textsc{Verified Return}\ \frac{\Gamma, pc \vdash_e e \hookrightarrow e' : \dot\tau \qquad \dot\tau^{pc} \preceq \dot\tau_r.\mathsf{ret} \qquad pc \preceq \dot\tau_r.\mathsf{wef}}{\Gamma, pc, \dot\tau_r \vdash_s \mathsf{return}\, e \hookrightarrow \mathsf{return}\, e'}$$

$$\textsc{Monitored Return}\ \frac{e \in \mathcal{E}xpr_{\lightning} \qquad s = @monitor(\Gamma, pc, \dot\tau_r, \mathsf{return}\, e)}{\Gamma, pc, \dot\tau_r \vdash_s \mathsf{return}\, e \hookrightarrow s} \qquad \textsc{Sequence}\ \frac{\forall_{i=1,2}\Gamma, pc, \dot\tau_r \vdash_s s_i \hookrightarrow s_i'}{\Gamma, pc, \dot\tau_r \vdash_s s_1; s_2 \hookrightarrow s_1'; s_2'}$$

Figure 2 Static Typing Core JS Expressions

Static Membership Check Since the domain of an object can change at execution time and since programs can check if a given field is defined using the keyword in, the mere existence of a field may disclose secret information. The existence security levels declared in object security types serve to control this type of information flows. However, analogously to field projections, this rule needs to take into account the whole prototype chain of the inspected object, because the field whose existence is being checked may be defined anywhere in that prototype chain. To this end, we make use of the static function $\bar\pi(\dot\tau, f)$ that computes the *lub* between the existence levels of f and *__proto__* in the prototype chain of objects of type $\dot\tau$.

▶ **Example 4** (Leaks via Membership Checks). The program below creates an object with two fields `secret1` and `secret2`. Then, depending on the value of a *high* variable `secret`, it deletes either `secret1` or `secret2` from the domain of `o`. Finally, the *low* variable `public` is assigned to `true` if `secret1` is defined in the prototype chain of `o` or to `false` if it is not, thereby creating an implicit flow between `secret` and `public`.

```
o = {}; o.secret1 = 0; o.secret2 = 0;
if (secret) { delete o.secret1 } else { delete o.secret2 }; public = secret1 in o
```

Listing 3 Security Leak via Field Membership Check

Letting Γ be the typing environment of Table 3, it follows that $\bar{\pi}(\Gamma(\mathtt{o}), \mathtt{secret1}) = \textsc{prim}^H$ because $\Gamma(\mathtt{o}).\overline{\mathtt{secret1}} = H$. Hence, the last assignment is not typable as the type of the expression `secret1 in o`, $\textsc{prim}^H$, is not lower than or equal to $\textsc{prim}^L$.

Static field assignment The first constraint of the rule checks if the type of the assigned expression is a subtype of the assigned field type, thus preventing the assignment of a secret value to a public field. The second constraint checks if the context level is lower than or equal to the existence level of the assigned field, thereby preventing the creation of a visible field depending on secret information.

Field Deletion The rule checks if the context level is lower than or equal to the field's existence level, thereby preventing visible fields from being deleted in invisible contexts.

Functional literal This rule checks if the context level is lower than or equal to the writing effect of the type of the function literal, thereby preventing the evaluation of function literals that update or create *public* resources inside secret contexts. Then, the type system types the body of the function literal using the typing environment obtained by extending the current one with the type of the the formal argument, the type of the keyword this, and the types of the variables declared in the body of the function literal. To this end, we make use of a syntactic function hoist that extends the typing environment given as its first argument with the mappings from the variables declared in the statement given as its second argument to their respective security types. Note that this rule may re-write the the body of the function literal in order to enable the dynamic analysis.

Function call This rule first verifies if the context level is lower than or equal to the writing effect of the function to call, thereby preventing the calling of a function that creates or updates *public* resources depending on *secret* values. Then, the rule checks if the type of the global object and the type of the function argument match the type of the keyword this and the type of the formal parameter. (Note that during the execution of a function call the keyword this is bound to the global object.) The function call is finally typed with the return type of the function type upgraded with the context level. Typing *static method calls* is similar to typing of function calls with the difference that the type system first determines the type of the method to apply.

Dyn. expression statement This rule wraps every expression that contains a dynamic field operation inside a boundary statement.

Conditional If the conditional guard contains a dynamic field operation, the whole conditional is wrapped inside a boundary statement. In the opposite case, the type system types both branches, upgrading the context level with the external level of the security type of the conditional guard.

Return If the returned expression contains a dynamic field operation, the type system wraps the whole return statement in a boundary statement. In the opposite case, the type system types the returned expression and checks that its type matches the security type of returned values. Furthermore, it must also be the case the current context level is lower than or equal to the writing effect of the function whose body is being typed. This restriction prevents a function from returning inside a secret context in some executions, while in others it does not return and writes low memory afterwards.

$$\textsc{Monitor sync}\quad \frac{\Psi \xrightarrow{\alpha_l} \Psi' \quad \Omega_i \xrightarrow{\alpha_l} \Omega_i' \quad \forall_{0\leq j\leq n, j\neq i}\, \Omega_j' = \Omega_j}{(\!|\Psi, \{\Omega_0, \cdots, \Omega_n\}|\!) \rightarrow (\!|\Psi', \{\Omega_0', \cdots, \Omega_n'\}|\!)}$$

$$\textsc{Unmonitored step}\quad \frac{\Psi \xrightarrow{\alpha_l} \Psi' \quad \forall_{0\leq j\leq n}\, \mathsf{er}(\Omega_i) \neq l}{(\!|\Psi, \{\Omega_0, \cdots, \Omega_n\}|\!) \rightarrow (\!|\Psi', \{\Omega_0, \cdots, \Omega_n\}|\!)}$$

$$\textsc{Monitor configuration} + \quad \frac{l = \mathsf{head}(L) \qquad \forall_{0\leq j\leq n}\, \mathsf{er}(\Omega_j) \neq l \qquad \Omega = \langle \Gamma, \dot{\tau}_r, l, pc :: [\,], [\,]\rangle}{(\!|\langle H, L, E[@monitor(\Gamma, \dot{\tau}_r, pc, s)]\rangle, \{\Omega_0, \cdots, \Omega_n\}|\!) \rightarrow (\!|\langle H, L, E[@monitor(s)]\rangle, \{\Omega_0, \cdots, \Omega_n, \Omega\}|\!)}$$

$$\textsc{Monitor configuration - 1}\quad \frac{\Psi = \langle H, L, E[@monitor(\underline{v})]\rangle \quad \mathsf{head}(L) = \mathsf{er}(\Omega) \quad \Psi' = \langle H, L, E[\underline{v}]\rangle}{(\!|\Psi, \{\Omega_0, \cdots, \Omega_n\} \cup \{\Omega\}|\!) \rightarrow (\!|\Psi', \{\Omega_0, \cdots, \Omega_n\}|\!)}$$

$$\textsc{Monitor configuration - 2}\quad \frac{\Psi = \langle H, L, E[@monitor(\mathsf{return}\ \underline{v})]\rangle \quad \mathsf{head}(L) = \mathsf{er}(\Omega) \quad \Psi' = \langle H, L, E[\mathsf{return}\ \underline{v}]\rangle}{(\!|\Psi, \{\Omega_0, \cdots, \Omega_n\} \cup \{\Omega\}|\!) \rightarrow (\!|\Psi', \{\Omega_0, \cdots, \Omega_n\}|\!)}$$

Figure 3 Monitored Semantics Rules

▶ **Example 5** (Hybrid versus Static Typing of the Running Example)**.** Consider the program from Example 1 and the typing environment of Table 3. When typing the assignment `public = o[g()]`, which contains a dynamic field operation, the type system applies the Dyn. expression statement rule and wraps the whole assignment inside a boundary statement. All the other statements, which do not contain dynamic field operations, are fully statically verified and, therefore, left unchanged. Hence, the resulting program is given by:

```
o = {}; o.secret1 = secret_input(); o.public1 = public_input();
o.public2 = public_input(); @monitor(@type_env, @pc, @ret, public = o[g()])
```

If, instead, the type system tried to statically type this assignment, it would need to check that the type of `o[g()]` was less than or equal to the type of `public`, $\textsf{PRIM}^L$. Since we do not know the value to which the call to `g` evaluates, the type system would need to use the *lub* between the types of all the fields declared in the type of o. Consequently, as one of those fields has type $\textsf{PRIM}^H$, the assignment would not be typable.

4 Dynamic Typing Secure Information Flow in Core JS

The goal of a boundary statement is to enable and disable the information flow analysis at runtime. In this section, we define the semantics of the boundary operator by extending the semantics of Core JS with optional tracking of security types and verification of security constraints.

Monitored Core JS semantics A configuration of the monitored semantics has the form $(\!|\Psi, \{\Omega_0, \cdots, \Omega_n\}|\!)$ where Ψ is a Core JS configuration as defined in Section 2 and $\{\Omega_0, \cdots, \Omega_n\}$ is a possibly empty set of monitor configurations. Each monitor configuration Ω is associated to a specific function call and has the form $\langle \Gamma, \dot{\tau}_r, l, o, \rho\rangle$ where: **(1)** Γ is a typing environment, **(2)** $\dot{\tau}_r$ is the type of the function that is executing, **(3)** l is the identifier of the environment record associated to the function call that is being monitored, **(4)** o is a *control context*, which is a list containing the levels of the expressions on which the monitored statement branched in order to reach the current context, and **(5)** ρ is an *expression context*, which is a list consisting of the security types of the values of the current evaluation context. The rules of the monitored semantics are given in Figure 3 and described below. We use $\mathsf{er}(\Omega)$ to denote the location of the environment record associated with Ω.

Rule Monitor sync corresponds to a monitored step. The transition of the monitor is synchronised with the transition of Core JS semantics through an *internal event* α_l, where

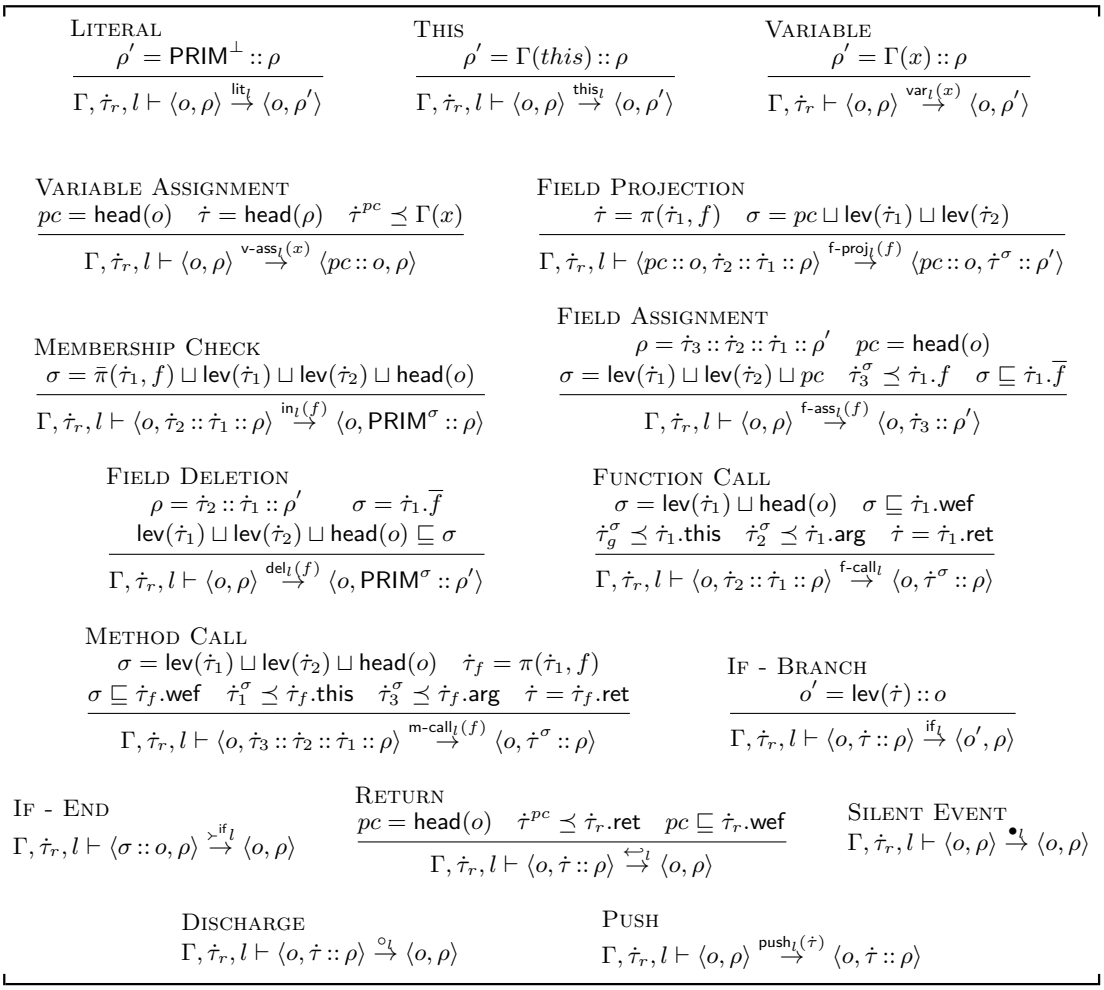

Figure 4 Dynamic Typing Core JS Expressions and Statements

l identifies the running function that performed a computation step.

Rule Unmonitored step models the case where there is no matching monitor configuration for the current computation step. In this case, Core JS semantics performs an unconstrained computation step (that takes place outside a boundary statement).

Rule Monitor configuration + generates a new monitor configuration for verifying the statement inside a boundary statement. In order to account for computation steps inside boundary statements, we extend the syntax of evaluation contexts with a special boundary context: $E = @monitor(E')$.

Rules Monitor configuration - 1 and Monitor configuration - 2 remove a monitor configuration from the current set of monitor configurations when its corresponding statement finishes executing.

Transitions between monitor configurations Monitor transitions are defined in Figure 4. We use $\Gamma, \dot\tau_r, l \vdash \langle o, \rho\rangle \overset{\alpha_l}{\to} \langle o', \rho'\rangle$ as a shorthand for $\langle \Gamma, \dot\tau_r, l, o, \rho\rangle \overset{\alpha_l}{\to} \langle \Gamma, \dot\tau_r, l, o', \rho'\rangle$. The constraints enforced by the monitor are the same as the constraints enforced by the type system of Section 3. However, in contrast to the type system, the monitor can precisely type dynamic expressions, because it has access to the field name computed at runtime. In fact, the internal event generated by all computations which involve inspecting the field of

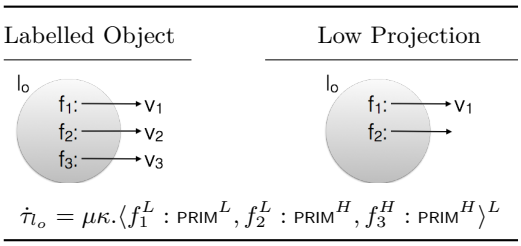

Figure 5 A Labelled Object and Its Low Projection

an object contains the name of the inspected field.

▶ **Example 6** (Monitoring a Dynamic Field Look-up)**.** We present the sequence of monitor configurations generated by the execution of `@monitor(@type_env, @pc, @ret, public = o[g()])` of our running example (when using the typing environment given in Figure 3).

$$\langle \bot, [\,] \rangle \xrightarrow{\text{var}_l(\text{o})} \langle L, \dot{\tau}_o \rangle \xrightarrow{\text{var}_l(\text{g})} \langle L, \langle \dot{\tau}_g . \_ \xrightarrow{H} \text{PRIM}^L \rangle^L :: \dot{\tau}_o \rangle \xrightarrow{\text{f-call}_l} \langle L, \text{PRIM}^L :: \dot{\tau}_o \rangle$$

(if `g()` returns `public1`)

$$\xrightarrow{\text{f-proj}_l(\texttt{public1})} \langle L, \text{PRIM}^L \rangle \xrightarrow{\text{v-ass}_l(\texttt{public})} \langle L, \text{PRIM}^L \rangle$$

(if `g()` returns `private1`)

$$\xrightarrow{\text{f-proj}_l(\texttt{private1})} \langle L, \text{PRIM}^H \rangle \xrightarrow{\text{v-ass}_l(\texttt{public})} \not\to$$

We consider two different cases: the case in which `g` evaluates to `public1` and the case in which it evaluates to `secret1`. While in the first case, the execution is allowed to go through, in the second one it gets stuck, because the program tries to assign a secret value to a public variable.

Let us now briefly explain the rules that better illustrate our choices when designing the monitor. Since, by default, all literal values are public, when a *literal* value is evaluated, the monitor simply pushes $\text{PRIM}^\bot$ onto the expression stack. In contrast, when a *variable* is evaluated, the monitor has to read its type from the typing environment and push it onto the expression stack. When a *field projection* is evaluated, the first two types on the expression stack are the types of the expressions that evaluate to the field name and to the inspected object, respectively. Furthermore, the name of the inspected field is available in the internal event that labels the transition. Hence, the monitor simply has to replace the first two types of the expression stack with the type of the inspected field upgraded with the external levels of the types of the current subexpressions. When an *if statement* is evaluated, the type of the conditional guard is on top of the expression stack. Hence, the monitor simply pops that type out of the expression stack and pushes its external level (upgraded with the current *pc*) onto the control stack. Complementarily, when the execution leaves the branch of a conditional, the monitor just pops out the top of the control stack.

Implementation. Instead of wrapping statements containing dynamic field operations within boundary statements, which are not part of the JavaScript language, the prototype of the hybrid type system [16] in-lines the monitoring logic in the statement itself [17]. This approach has the advantage of being immediately deployable. This prototype implementation was used to verify a simple Web application described in appendix.

5 Security Guarantees

This section describes the security guarantees offered by the proposed analysis. To formally define the absence of information leaks, we rely on an intuitive notion of *low-projection* [15] that establishes the part of a heap that an attacker at a given security level can see. Informally, given a heap H, an attacker at level σ can observe:

1. the existence of a field f in the domain of an object whose type has external level $\leq \sigma$ and associates f with an existence level $\leq \sigma$ and
2. the value of a field f in the domain of an object whose type has external level $\leq \sigma$ and associates f with a security type with external level $\leq \sigma$.

Figure 5 presents a labelled object together with its low-projection at level L. The object in the figure has three fields: f_1, f_2, and f_3. An attacker at level L can observe both the existence and the value of f_1 since it has *low* existence level and is associated with a visible value and the existence but not the value of f_2, since it has *low* existence level but is associated with an invisible value. The attacker can neither observe the value nor the existence of f_3 because it has *high* existence level and is associated with an invisible value. Two heaps H_0 and H_1 are said to be *low-equal* at level σ, written $H_0 \sim_\sigma H_1$ if they coincide in their respective low-projections. Theorem 21 states that the monitored successfully-terminating execution of a program generated by the static type system on two low-equal heaps always yields two low-equal heaps. A sketch of the proof of Theorem 21 is given in appendix.

▶ **Theorem 7** (Noninterference). *For any typing environment Γ, levels σ and pc, security type $\dot{\tau}$, statement, s, and two heaps H_0 and H_1, such that $\Gamma, pc, \dot{\tau} \vdash_s s \hookrightarrow s'$, $H_0 \sim_\sigma H_1$, and $(\!|\langle H_i, [\,], s'\rangle, \{\}|\!) \rightarrow^* (\!|\langle H_i', [\,], v_i\rangle, \{\}|\!)$ for $i = 0, 1$, it holds that $H_0' \sim_\sigma H_1'$.*

6 Related Work

There is a wide variety of mechanisms for enforcing and verifying secure information flow, ranging from purely static type systems [19, 15] to different flavours of dynamic analysis [14, 3]. The main mechanisms for securing information flow in JavaScript [1, 9, 7] are mostly-dynamic due to the dynamic language nature.

There is a long line of research on safety types for JavaScript which dates back to the seminal work of Thieman [18]. Since then, the TypeScript programming language [12] was proposed as a flexible language that adds optional types to JavaScript with the goal of harnessing the flexibility of real JavaScript, while at the same time providing some of the advantages otherwise reserved for statically typed languages, such as informative compiling errors. Recently, Rastogi et al. [13] designed and implemented a new gradual type system for safely compiling TypeScript to JavaScript. The soundness of the proposed approach is guaranteed by combining strict static checks with residual runtime checks. We believe that our work can serve as a basis for extending TypeScript types with security labels in order to verify secure information flow in TypeScript web applications.

Gradual type systems for secure information flow have been proposed for a pure lambda calculus [4] and for a core ML-like language with references [5]. The goal of these two works is significantly different from ours, as their main intent is to cater for the use of *polymorphic* security labels. For instance, the type language proposed in [5] includes a special annotation "?" representing an unknown security level at static time. Expressions that use variables whose types contain the unknown level annotation, "?", cannot be precisely typed at static time. The programmer can introduce runtime casts in points where values of a predetermined security type are expected. Then the dynamic analysis checks whether or not a cast can be *securely* performed during execution. However, in order to verify such casts at runtime, these analyses must track security labels during the execution of both dynamically verified and statically verified program regions. In contrast, our analysis only needs to dynamically verify the execution of program regions which were not statically verified.

7 Conclusions

We propose a sound hybrid typing analysis for enforcing secure information flow in a core of JavaScript that includes dynamic field operations. Furthermore, our analysis can be easily extended to handle other dynamic constructs of the language such as eval or unknown code, which only need to be wrapped inside the proposed boundary statement. Finally, we have implemented our analysis and used it to verify a web application described in the appendix.

This work follows a well-established trend on combining static and dynamic analysis to devise more permissive and efficient hybrid mechanisms [14]. Our approach can be applied to other scenarios, such as the verification of isolation properties [10], where it could be used to derive mostly-static lightweight enforcement mechanisms from prior purely static specifications.

References

1 A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, 2014.
2 R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, 2009.
3 D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *SP*, 2010.
4 T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.
5 L. Fennell and P. Thiemann. Gradual security typing with references. In *CSF*, 2013.
6 C. Flanagan. Hybrid type checking. In *POPL*, 2006.
7 W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *CCS*, 2012.
8 D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.
9 D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF*, 2012.
10 S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *CSF*, 2009.
11 J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM TOPLAS*, 2009.
12 Microsoft. TypeScript language specification. Technical report, Microsoft, 2014.
13 A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, 2015.
14 A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.
15 A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
16 J. Fragoso Santos. Online materials - hybrid type system. http://www.doc.ic.ac.uk/~jfaustin, 2015.
17 J. Fragoso Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *IFIP SEC*, 2014.
18 P. Thiemann. Towards a type system for analysing JavaScript programs. In *ESOP*, 2005.
19 D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 1996.
20 A. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 1994.

A Auxiliary Definitions

This appendix formally introduces all the definitions omitted in the paper. In particular, the full semantics is given in Figure 6.

A.1 Semantics

Scope resolution: $\sigma(H, L, x)$

$$\frac{(l,x) \in \mathsf{dom}(H)}{\sigma(H, l :: L, x) \triangleq l} \qquad \frac{(l,x) \notin \mathsf{dom}(H)}{\sigma(H, l :: L, x) \triangleq \sigma(H, L, x)} \qquad \frac{\pi(H, l_g, x) = l'}{\sigma(H, [\,], x) \triangleq l'}$$

Prototype resolution: $\pi(H, l, x)$

$$\pi(H, \mathsf{null}, x) \triangleq \mathsf{null} \qquad \frac{(l,x) \in \mathsf{dom}(H)}{\pi(H, l, x) \triangleq l} \qquad \frac{(l,x) \notin \mathsf{dom}(H) \qquad l' = H(l, \_proto\_)}{\pi(H, l, x) \triangleq \pi(H, l', x)}$$

Environment Records: act

$$\frac{\mathsf{defs}(s) = \{y_1, \cdots, y_m\} \qquad H = l \mapsto \{x : v, @this : l', y_1 : \mathsf{undefined}, \cdots, y_m : \mathsf{undefined}\}}{\mathsf{act}(l, x, v, s, l') \triangleq \uplus H}$$

A.2 Static Typing

Type Field Projection

$$\frac{\dot\tau = \mu\kappa.\langle \cdots, f^{\sigma_i} : \dot\tau_i, \cdots \rangle^{\sigma}}{(\dot\tau.f, \dot\tau.\overline{f}) \triangleq (\{\dot\tau/\kappa\}\dot\tau_i, \sigma_i)} \text{ Defined Field} \qquad \frac{\dot\tau = \mu\kappa.\langle \cdots, *^{\sigma_*} : \dot\tau_* \rangle^{\sigma} \qquad f \notin \mathsf{dom}(\dot\tau)}{(\dot\tau.f, \dot\tau.\overline{f}) \triangleq (\{\dot\tau/\kappa\}\dot\tau_*, \sigma_*)} \text{ Non Defined Field}$$

Type Field Look Up - Security Type

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \sigma = \dot\tau.\overline{\_proto\_} \quad \dot\tau' = \pi(\dot\tau.\_proto\_, f) \neq \bot}{\pi(\dot\tau, f) \triangleq (\dot\tau' \curlyvee \dot\tau.f)^{\sigma}} \text{ Defined Field with Proto - 1}$$

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \pi(\dot\tau.\_proto\_, f) = \varnothing}{\pi(\dot\tau, f) \triangleq \dot\tau.f} \text{ Defined Field with Proto - 2}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \sigma = \dot\tau.\overline{\_proto\_} \quad \dot\tau' = \pi(\dot\tau.\_proto\_, f) \neq \varnothing}{\pi(\dot\tau, f) \triangleq (\dot\tau')^{\sigma}} \text{ Undefined Field with Proto - 1}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \pi(\dot\tau.\_proto\_, f) = \varnothing}{\pi(\dot\tau, f) \triangleq \varnothing} \text{ Undefined Field with Proto - 2}$$

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \notin \mathsf{dom}(\dot\tau)}{\pi(\dot\tau, f) \triangleq \dot\tau.f} \text{ Defined Field with No Proto}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \notin \mathsf{dom}(\dot\tau)}{\pi(\dot\tau, f) \triangleq \varnothing} \text{ Undefined Field with No Proto}$$

LITERAL
$$\frac{l = \mathsf{head}(L)}{\langle H, L, lit\rangle \xrightarrow{\mathsf{lit}_l} \langle H, L, \underline{lit}\rangle}$$

THIS
$$\frac{l = \mathsf{head}(L) \qquad l' = H(l, @this)}{\langle H, L, \mathsf{this}\rangle \xrightarrow{\mathsf{this}_l} \langle H, L, l'\rangle}$$

VARIABLE
$$\frac{l' = \sigma(H, l :: L, x) \qquad \underline{v} = H(l', x)}{\langle H, l :: L, x\rangle \xrightarrow{\mathsf{var}_l(x)} \langle H, l :: L, \underline{v}\rangle}$$

ASSIGNMENT
$$\frac{l = \mathsf{head}(L) \qquad l' = \sigma(H, L, x) \qquad H' = H[l'.x \mapsto \underline{v}]}{\langle H, L, x = \underline{v}\rangle \xrightarrow{\text{v-ass}_l(x)} \langle H', L, \underline{v}\rangle}$$

OBJECT LITERAL
$$\frac{l = \mathsf{head}(L) \qquad l' = \mathsf{fresh}(H, \dot{\tau}) \qquad H' = H \uplus (l', \_proto\_) \mapsto l_{op}}{\langle H, L, \{\,\}\,[\dot{\tau}]\rangle \xrightarrow{\mathsf{push}_l(\dot{\tau})} \langle H', L, l'\rangle}$$

STATIC FIELD PROJECTION
$$\frac{l' = \mathsf{head}(L)}{\langle H, L, l.f\rangle \xrightarrow{\mathsf{lit}_{l'}} \langle H, L, l[f]\rangle}$$

DYNAMIC FIELD PROJECTION
$$\frac{l'' = \pi(H, l', f) \qquad \underline{v} = H(l'', f)}{\langle H, l :: L, l'[f]\rangle \xrightarrow{\mathsf{f\text{-}proj}_l(f)} \langle H, l :: L, \underline{v}\rangle}$$

DYNAMIC FIELD ASSIGNMENT
$$\frac{l' = \mathsf{head}(L) \qquad H' = H[l.f \mapsto \underline{v}]}{\langle H, L, l[f] = \underline{v}\rangle \xrightarrow{\mathsf{f\text{-}ass}_{l'}(f)} \langle H', L, \underline{v}\rangle}$$

DYNAMIC FIELD ASSIGNEMENT
$$\frac{l' = \mathsf{head}(L)}{\langle H, L, l[x] = e\rangle \xrightarrow{\bullet_{l'}} \langle H, L, l.x = e\rangle}$$

MEMBERSHIP CHECK - FALSE
$$\frac{l' = \mathsf{head}(L) \qquad \pi(H, l, f) = \mathsf{null}}{\langle H, L, [f]\,\mathsf{in}\,l\rangle \xrightarrow{\mathsf{in}_{l'}(f)} \langle H, L, \mathsf{false}\rangle}$$

MEMBERSHIP CHECK - TRUE
$$\frac{l' = \mathsf{head}(L) \qquad \pi(H, l, f) \neq \mathsf{null}}{\langle H, L, [f]\,\mathsf{in}\,l\rangle \xrightarrow{\mathsf{in}_{l'}(f)} \langle H, L, \mathsf{true}\rangle}$$

STATIC MEMBERSHIP CHECK
$$\frac{l' = \mathsf{head}(L)}{\langle H, L, [f]\,\mathsf{in}\,l\rangle \xrightarrow{\bullet_{l'}} \langle H, L, f\,\mathsf{in}\,l\rangle}$$

DELETE - FALSE DELETE
$$\frac{l' = \mathsf{head}(L) \qquad (l, f) \notin \mathsf{dom}(H)}{\langle H, L, \mathsf{delete}\,l[f]\rangle \xrightarrow{\mathsf{del}_{l'}(f)} \langle H, L, \mathsf{false}\rangle}$$

DELETE - TRUE
$$\frac{l' = \mathsf{head}(L) \qquad H = H' \uplus (l, f) \mapsto \underline{v}}{\langle H, L, \mathsf{delete}\,l[f]\rangle \xrightarrow{\mathsf{del}_{l'}(f)} \langle H', L, \mathsf{true}\rangle}$$

STATIC DELETE
$$\frac{l' = \mathsf{head}(L)}{\langle H, L, \mathsf{delete}\,l.x\rangle \xrightarrow{\bullet_{l'}} \langle H, L, \mathsf{delete}\,l.x\rangle}$$

FUNCTION LITERAL
$$\frac{l' = \mathsf{fresh}(H, \dot{\tau}) \qquad H' = H \uplus l' \mapsto \{@scope : l :: L, @body : \lambda x : \dot{\tau}.s\}}{\langle H, l :: L, \mathsf{function}\,(x)[\dot{\tau}]\{s\}\rangle \xrightarrow{\mathsf{push}_l(\dot{\tau})} \langle H', l :: L, l'\rangle}$$

FUNCTION CALL
$$\frac{\begin{array}{c} l' = \mathsf{head}(L) \qquad l'' \notin \mathsf{dom}(H) \\ \lambda x : \dot{\tau}.s = H(l, @body) \qquad L' = H(l, @scope) \\ H' = H \uplus \mathsf{act}(l'', x, \underline{v}, s, l_g) \end{array}}{\langle H, L, l(\underline{v})\rangle \xrightarrow{\mathsf{f\text{-}call}_{l'}} \langle H', l'' :: L', @\mathsf{FunExe}(L, s)\rangle}$$

DYNAMIC METHOD CALL
$$\frac{\begin{array}{c} l' = \mathsf{head}(L) \qquad l'' = \pi(H, l, f) \qquad l_f = H(l'', f) \\ \lambda x : \dot{\tau}.s = H(l_f, @body) \qquad L' = H(l_f, @scope) \\ l''' \notin \mathsf{dom}(H) \qquad H' = H \uplus \mathsf{act}(l''', x, \underline{v}, s, l) \end{array}}{\langle H, L, l[f](\underline{v})\rangle \xrightarrow{\mathsf{m\text{-}call}_{l'}(x)} \langle H', l''' :: L', @\mathsf{FunExe}(L, s)\rangle}$$

DYNAMIC METHOD CALL
$$\frac{l' = \mathsf{head}(L)}{\langle H, L, l[x](\underline{v})\rangle \xrightarrow{\bullet_{l'}} \langle H, L, l.x(\underline{v})\rangle}$$

RETURN UNDEFINED
$$\langle H, L, @\mathsf{FunExe}(L', \underline{v})\rangle \xrightarrow{\succ_l} \langle H, L', \mathsf{undefined}\rangle$$

VAR DECLARATION
$$\frac{l = \mathsf{head}(L)}{\langle H, L, \mathsf{var}\,x\,[\dot{\tau}]\rangle \xrightarrow{\bullet_l} \langle H, L, \mathsf{undefined}\rangle}$$

SEQUENCE
$$\frac{l = \mathsf{head}(L)}{\langle H, L, \underline{v}; s\rangle \xrightarrow{\circ_l} \langle H, L, s\rangle}$$

SEQUENCE - RETURN
$$\frac{l = \mathsf{head}(L)}{\langle H, L, \mathsf{return}\,\underline{v}; s\rangle \xrightarrow{\bullet_l} \langle H, L, \mathsf{return}\,\underline{v}\rangle}$$

RETURN VALUE
$$\frac{l = \mathsf{head}(L)}{\langle H, L, @\mathsf{FunExe}(L', \mathsf{return}\,\underline{v})\rangle \xrightarrow{\hookleftarrow_l} \langle H, L', \underline{v}\rangle}$$

IF - FALSE
$$\frac{l = \mathsf{head}(L) \qquad \mathsf{false}(\underline{v}) \qquad s' = s_2; @\mathsf{EndIf}}{\langle H, L, \mathsf{if}(\underline{v})\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \xrightarrow{\mathsf{if}_l} \langle H, L, s'\rangle}$$

IF - TRUE
$$\frac{l = \mathsf{head}(L) \qquad \neg\mathsf{false}(\underline{v}) \qquad s' = s_1; @\mathsf{EndIf}}{\langle H, L, \mathsf{if}(\underline{v})\,\{s_1\}\,\mathsf{else}\,\{s_2\}\rangle \xrightarrow{\mathsf{if}_l} \langle H, L, s'\rangle}$$

IF END
$$\frac{l = \mathsf{head}(L)}{\langle H, L, @\mathsf{EndIf}\rangle \xrightarrow{\succ_l} \langle H, L, \mathsf{undefined}\rangle}$$

CONTEXTUAL PROPAGATION
$$\frac{\langle H, L, s\rangle \xrightarrow{\alpha} \langle H', L', s'\rangle}{\langle H, L, E[s]\rangle \xrightarrow{\alpha} \langle H', L', E[s']\rangle}$$

Figure 6 Small-Step Semantics of Core JS

Type Field Look Up - Existence Level

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \sigma = \dot\tau.\overline{\_proto\_} \quad \sigma' = \bar\pi(\dot\tau.\_proto\_, f) \neq \varnothing}{\bar\pi(\dot\tau, f) \triangleq \dot\tau.\overline{f} \sqcup \sigma \sqcup \sigma'} \text{ Defined Field with Proto - 1}$$

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \bar\pi(\dot\tau.\_proto\_, f) = \varnothing}{\bar\pi(\dot\tau, f) \triangleq \dot\tau.\overline{f}} \text{ Defined Field with Proto - 2}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \sigma = \dot\tau.\overline{\_proto\_} \quad \sigma' = \bar\pi(\dot\tau.\_proto\_, f) \neq \varnothing}{\pi(\dot\tau, f) \triangleq \sigma \sqcup \sigma'} \text{ Undefined Field with Proto - 1}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \in \mathsf{dom}(\dot\tau) \quad \bar\pi(\dot\tau.\_proto\_, f) = \varnothing}{\pi(\dot\tau, f) \triangleq \varnothing} \text{ Undefined Field with Proto - 2}$$

$$\frac{f \in \mathsf{dom}(\dot\tau) \vee * \in \mathsf{dom}(\dot\tau) \quad \_proto\_ \notin \mathsf{dom}(\dot\tau)}{\bar\pi(\dot\tau, f) \triangleq \dot\tau.\overline{f}} \text{ Defined Field with No Proto}$$

$$\frac{f \notin \mathsf{dom}(\dot\tau) \quad * \notin \mathsf{dom}(\dot\tau) \quad \_proto\_ \notin \mathsf{dom}(\dot\tau)}{\bar\pi(\dot\tau, f) \triangleq \varnothing} \text{ Undefined Field with No Proto}$$

▶ **Definition 8** (Hoisting a Variable Environment)**.** Given a typing environment Γ and a statement s, we define $\mathsf{hoist}(\Gamma, s)$ inductively as follows:

$$\mathsf{hoist}(\Gamma, s) \triangleq \begin{cases} \Gamma & \text{if } s \in \mathcal{E}xpr \\ \Gamma[x \mapsto \dot\tau] & \text{if } s = \mathsf{var}\, x\, [\dot\tau] \\ \mathsf{hoist}(\mathsf{hoist}(\Gamma, s_1), s_2) & \text{if } s = s_1; s_2 \\ \mathsf{hoist}(\mathsf{hoist}(\Gamma, s_1), s_2) & \text{if } s = \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \\ \Gamma & \text{if } s = \mathsf{return}\, e \end{cases}$$

A.3 Low-Equality for Lists

Informally, 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 the levels that are associated with both of them are $\not\sqsubseteq \sigma$. Definition 9 formalizes this notion.

▶ **Definition 9** (Low-Equality for Lists)**.** Two lists of values $\overrightarrow{v}$ and $\overrightarrow{v}'$ respectively labeled by two lists of security levels $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ are said to be low-equal w.r.t. a security level σ, written $\overrightarrow{v}, \overrightarrow{\sigma} \sim_\sigma \overrightarrow{v}', \overrightarrow{\sigma}'$ if the following hold: **((1))** $\forall_{0 \leq i < n}\ \overrightarrow{\sigma}(i) \sqcap \overrightarrow{\sigma}'(i) \sqsubseteq \sigma \Rightarrow \overrightarrow{v}(i) = \overrightarrow{v}'(i) \wedge \overrightarrow{\sigma}(i) = \overrightarrow{\sigma}'(i) \sqsubseteq \sigma$, **((2))** $\forall_{n < i < |\overrightarrow{\sigma}|}\ \overrightarrow{\sigma}(i) \not\sqsubseteq \sigma$, and **((3))** $\forall_{n < i < |\overrightarrow{\sigma}'|}\ \overrightarrow{\sigma}'(i) \not\sqsubseteq \sigma$, where $n = \mathsf{min}(|\overrightarrow{\sigma}|, |\overrightarrow{\sigma}'|)$.

Given a list of security levels used as a control context $\overrightarrow{\sigma}$, we define the low-projection of $\overrightarrow{\sigma}$ w.r.t. a given security level σ, written $\overrightarrow{\sigma}\restriction^\sigma$ as the prefix of $\overrightarrow{\sigma}$ that only contains levels $\sqsubseteq \sigma$.

A.4 Low Equality for Memories and Scope

▶ **Definition 10** (Low-Projection and Low-Equal Scope Chains)**.** The low-projection of a scope chain L stored in a heap H at a security level σ wrt a typing environment Γ is given by:

$$(H, L)\restriction^{\Gamma,\sigma} = \{(v, x) \,|\, H(l_x, x) = v \wedge \mathsf{lev}(\Gamma(x)) \sqsubseteq \sigma \wedge \sigma(H, L, x) = l_x\}$$

Two scope chains (H_0, L_0) and (H_1, L_1) are said to be *low-equal* at level σ, written $\Gamma \vdash H_0, L_0 \sim_\sigma H_1, L_1$ if they coincide in their respective low-projections, $(H_0, L_0)\restriction^{\Gamma,\sigma} = (H_1, L_1)\restriction^{\Gamma,\sigma}$.

▶ **Definition 11** (Low-Projection and Low-Equality for Heaps)**.** The low-projection of a heap H at a security level σ is given by:

$$\begin{array}{rcl} H\restriction^{\sigma} & = & \{(l_{\dot\tau}, f) \,|\, (l_{\dot\tau}, f) \in \mathsf{dom}(H) \wedge \dot\tau.\overline{f} \sqcup \mathsf{lev}(\dot\tau) \sqsubseteq \sigma\} \\ & & \cup \{(l_{\dot\tau}, f, v) \,|\, (l_{\dot\tau}, f) \in \mathsf{dom}(H) \wedge H(l_{\dot\tau}, f) = v \wedge \mathsf{lev}(\dot\tau.f) \sqcup \mathsf{lev}(\dot\tau) \sqsubseteq \sigma\} \end{array}$$

Two heaps H_0 and H_1 are said to be *low-equal* at level σ, written $H_0 \sim_\sigma H_1$ if they coincide in their respective low-projections, $H_0\restriction^{\sigma} = H_1\restriction^{\sigma}$.

A.5 Low-Equality for Contexts

We introduce a new function $\mathsf{pvalues}$ that given a context, returns the list containing the values in that context that were already evaluated in *evaluation order*. Definition 12 formalizes this notion.

▶ **Definition 12** (Context Parsed Values)**.** Given a context E, we recursively define $\mathsf{pvalues}(E)$ as follows:

$$\begin{array}{lcllcl}
\mathsf{pvalues}(\square) & = & [\,] & \mathsf{pvalues}(x = \hat{E}) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\hat{E}.f) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\hat{E}[e]) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(l[\hat{E}]) & = & l :: \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\hat{E}.f = e) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\hat{E}[e_1] = e_2) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(l[\hat{E}] = e) & = & l :: \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(l[f] = \hat{E}) & = & f :: l :: \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}([\hat{E}]\,\mathsf{in}\,e) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}([f]\,\mathsf{in}\,\hat{E}) & = & f :: \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\mathsf{delete}\,\hat{E}.f) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\mathsf{delete}\,\hat{E}[e]) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\mathsf{delete}\,l[\hat{E}]) & = & l :: \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\hat{E}(e)) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(l(\hat{E})) & = & l :: \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\hat{E}.f(e)) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\hat{E}[e](e)) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(l[\hat{E}](e)) & = & l :: \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(l[f](\hat{E})) & = & f :: l :: \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(E;\, s) & = & \mathsf{pvalues}(E) & \mathsf{pvalues}(\mathsf{if}(\hat{E})\,\{s_1\}\,\mathsf{else}\,\{s_2\}) & = & \mathsf{pvalues}(\hat{E}) \\
\mathsf{pvalues}(\mathsf{return}\,\hat{E}) & = & \mathsf{pvalues}(\hat{E}) & \mathsf{pvalues}(\mathsf{if}(\hat{E})\,\{s_1\}\,\mathsf{else}\,\{s_2\}) & = & \mathsf{pvalues}(\hat{E})
\end{array}$$

We extend the notion of low-projection to contexts paired up with a list of security levels corresponding to the control context. Hence, given an execution context E and a control context $\overrightarrow{\sigma}$, $E\restriction^{\overrightarrow{\sigma},\sigma}$ denotes the context obtained from E by removing its subcontext that is not observable. For instance, $@\mathsf{EndIf}(@\mathsf{EndIf}(x = []); x = 1)\restriction^{L::H,L} = @\mathsf{EndIf}([]; x = 1)$.

▶ **Definition 13** (Context Low-Projection)**.** Given a context E and control context $\overrightarrow{\sigma}$, we define the low-projection of E at a level σ w.r.t. $\overrightarrow{\sigma}$ recursively as follows:

$$E\restriction^{\sigma' :: \overrightarrow{\sigma},\sigma} = \begin{cases} E & \text{if } \sigma' \sqsubseteq \sigma \\ \mathsf{flush}(E)\restriction^{\overrightarrow{\sigma},\sigma} & \text{otherwise} \end{cases}$$

where $\mathsf{flush}(E) = E'$ if and only if $E = E'[@\mathsf{EndIf}(E'')]$ and there are no contexts $\hat{E}$ and $\hat{E}'$ such that $E'' = \hat{E}[@\mathsf{EndIf}(\hat{E}')]$.

Using the definition of low-projection for contexts, we extend the definition of low-equality for contexts paired up with the corresponding control and expression contexts. In the following, we use: **((1))** $|E|$ as an abbreviation for $|\mathsf{pvalues}(E)|$, **((2))** $\lceil L \rceil_n$ for the list containing the first n elements of L, and **((3))** $\lfloor L \rfloor_n$ for the list containing the last n elements of L.

▶ **Definition 14** (Low-Equality for Contexts)**.** Given two contexts E_0 and E_1 each paired up with two lists of security levels $\overrightarrow{\sigma}_i$ and $\overrightarrow{\sigma}'_i$ for $i = 0, 1$ and a security level σ, we say that E_0 is low-equal to E_1 w.r.t. $\overrightarrow{\sigma}_0, \overrightarrow{\sigma}'_0, \overrightarrow{\sigma}_1, \overrightarrow{\sigma}'_1$ and σ, written $E_0, \overrightarrow{\sigma}_0, \overrightarrow{\sigma}'_0 \sim_\sigma E_1, \overrightarrow{\sigma}_1, \overrightarrow{\sigma}'_1$, if and only if: $\mathsf{pvalues}(E'_0), \lfloor \overrightarrow{\sigma}_0 \rfloor_{|E'_0|} \sim_\sigma \mathsf{pvalues}(E'_1), \lfloor \overrightarrow{\sigma}_1 \rfloor_{|E'_1|}$ where $E'_0 = E_0\restriction^{\overrightarrow{\sigma}_0,\sigma}$ and $E'_1 = E_1\restriction^{\overrightarrow{\sigma}_1,\sigma}$.

A.6 Low Equality for Redexes

Let us define a *redex* as a statement $\bar{s}$ for which there is no context E different from $\square$ and statement s such that $\bar{s} = E[s]$. Definition 15 extends $\mathsf{pvalues}$ to rexedes.

▶ **Definition 15** (Runtime Values of a Redex)**.** Given a redex $\bar{s}$, we define $\mathsf{pvalues}(\bar{s})$ as follows:

$\mathsf{pvalues}(v)$	$=$	$[\,]$	$\mathsf{pvalues}(\mathsf{this})$	$=$	$[\,]$
$\mathsf{pvalues}(x)$	$=$	$[\,]$	$\mathsf{pvalues}(x = \underline{v})$	$=$	$\underline{v}$
$\mathsf{pvalues}(\{\,\}\,[\,])$	$=$	$[\,]$	$\mathsf{pvalues}(l[f])$	$=$	$f :: l$
$\mathsf{pvalues}(l[f] = \underline{v})$	$=$	$\underline{v} :: f :: l$	$\mathsf{pvalues}([f]\,\mathsf{in}\,l)$	$=$	$f :: l$
$\mathsf{pvalues}(\mathsf{delete}\,l[f])$	$=$	$f :: l$	$\mathsf{pvalues}(\mathsf{function}\,(x)[\dot{\tau}]\{s\})$	$=$	$[\,]$
$\mathsf{pvalues}(l(\underline{v}))$	$=$	$\underline{v} :: l$	$\mathsf{pvalues}(l[f](\underline{v}))$	$=$	$\underline{v} :: f :: l$
$\mathsf{pvalues}(\mathsf{var}\,x\,[\dot{\tau}])$	$=$	$[\,]$	$\mathsf{pvalues}(\underline{v};\ s_2)$	$=$	$\underline{v}$
$\mathsf{pvalues}(\mathsf{if}(\underline{v})\,\{s_1\}\,\mathsf{else}\,\{s_2\})$	$=$	$\underline{v}$	$\mathsf{pvalues}(\mathsf{return}\,\underline{v})$	$=$	$\underline{v}$

We say that two redexes $\bar{s}$ and $\bar{s}'$ are *equal up to runtime-values*, written $\bar{s} \equiv \bar{s}'$, if they only differ in runtime values. We use $|\bar{s}|$ as an abbreviaton for $|\mathsf{pvalues}(\bar{s})|$. We extend the definition of low-equality to redexes, each paired up with a list of security levels in the following way. Two redexes $\bar{s}_0$ and $\bar{s}_1$ paired up with two lists of security levels $\overrightarrow{\sigma}_0$ and $\overrightarrow{\sigma}_1$ respectively are said to be low-equal at level σ, written $\bar{s}_0, \overrightarrow{\sigma}_0 \sim_\sigma \bar{s}_1, \overrightarrow{\sigma}_1$, if and only if $\bar{s}_0 \equiv \bar{s}_1$ and $\mathsf{pvalues}(\bar{s}_0), \overrightarrow{\sigma}_0 \sim_\sigma \mathsf{pvalues}(\bar{s}_1), \overrightarrow{\sigma}_1$.

A.7 Low Equality for Configurations of the Monitored Semantics

Definition 16 extends the notion of low-equality for JavaScript configurations respectively. An intermediate configuration is never low-equal to a final configuration.

▶ **Definition 16** (Low-Equality for Monitored Intermediate Confs)**.** Two JavaScript confs. $\langle H, L, E[\bar{s}]\rangle$, $\langle H', L', E'[\bar{s}']\rangle$ monitored by two monitor configurations $\langle \Gamma, pc, \dot{\tau}_r, o, \rho\rangle$ and $\langle \Gamma, pc, \dot{\tau}_r, o', \rho'\rangle$ are said to be low-equal at level σ, written $\Gamma, pc, \dot{\tau}_r \vdash \langle H, L, E[\bar{s}]\rangle, o, \rho \sim_\sigma \langle H', L', E'[\bar{s}']\rangle, o', \rho'$, if and only if:

1. $H \sim_\sigma H'$
2. $\Gamma \vdash H, L \sim_\sigma H', L'$
3. $E, o, \rho \sim_\sigma E', o', \rho'$
4. $o\restriction^\sigma = o'\restriction^\sigma$
5. $\mathsf{lev}(o) \sqcap \mathsf{lev}(o') \sqsubseteq \sigma \Rightarrow \mathsf{lev}(o) = \mathsf{lev}(o')\ \wedge\ \bar{s}, \lceil o\rceil_{|\bar{s}|} \sim_\sigma \bar{s}', \lceil o'\rceil_{|\bar{s}'|}$

B Case Study

This appendix presents a small Web application used to demonstrate the applicability of the type system. It consists of a simple contact manager online application, given in Listing 4. The variable `CM` holds the *Contact Manager* object. The contact manager stores contacts in an object that is bound to its field `contact_list`. This object 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 field `fst`), a last name (stored in field `lst`), an e-mail address (stored in field `email`), and a flag `favourite` (whose existence indicates that that contact is among the user's favourite contacts).

This example illustrates the typical use of prototype-based inheritance in JavaScript. We create a "fixed" object for storing all the methods contact objects must implement and we assign this object to the field `proto_contact` of the *Contact Manager*. 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 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.

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

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

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

CM.proto_contact.unFavourite = function() {
   if ("favourite" in this) {
      delete this.favourite
   } else {
      true
   }
};

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

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

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

CM.getContact = function(lst_name, i) {
```

$$\dot{\tau}_{contact} = \mu\kappa \cdot \left\langle \begin{array}{l} \texttt{fst}^L : \mathsf{PRIM}^L, \texttt{lst}^L : \mathsf{PRIM}^L, \\ \texttt{id}^L : \mathsf{PRIM}^H, \texttt{favourite}^H : \mathsf{PRIM}^H, \\ \texttt{\_proto\_}^L : \dot{\tau}_{proto\_contact} \end{array} \right\rangle^L$$

$$\dot{\tau}_{proto\_contact} = \mu\kappa \cdot \left\langle \begin{array}{l} \texttt{printContact}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^L \rangle^L, \\ \texttt{makeFavorite}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^L \rangle^L, \\ \texttt{isFavorite}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^H \rangle^L, \\ \texttt{unFavorite}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^H \rangle^L \end{array} \right\rangle^L$$

$$\dot{\tau}_{CM} = \mu\kappa \cdot \left\langle \begin{array}{l} \texttt{proto\_contact}^L : \dot{\tau}_{proto\_contact}, \\ \texttt{contact\_list}^L : \mu\kappa.\langle *^L : \dot{\tau}_{contact} \rangle^L, \\ \texttt{createContact}^L : \langle \kappa.(\mathsf{PRIM}^L, \mathsf{PRIM}^L, \mathsf{PRIM}^H) \xrightarrow{L} \dot{\tau}_{contact} \rangle^L, \\ \texttt{storeContact}^L : \langle \kappa.(\dot{\tau}_{contact}, \mathsf{PRIM}^L) \xrightarrow{L} \mathsf{PRIM}^L \rangle^L, \\ \texttt{getContact}^L : \langle \kappa.(\mathsf{PRIM}^L, \mathsf{PRIM}^L) \xrightarrow{H} \dot{\tau}_{contact} \rangle^L \end{array} \right\rangle^L$$

Figure 7 Typing Environment for the Contact Manager - $\Gamma_{CM} = [\texttt{CM} \mapsto \dot{\tau}_{CM}]$

```
    return this.contact_list[lst_name+i]
}
```

Listing 4 A Simple Contact Manager

In the following, we give a brief description of the methods that compose the Contact Manager example.

- **Methods of Contact Objects.** The method `printContact` returns a string consisting of the last and first names of the contact on which it was called separated by a comma (in this context, 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 an arbitrary value to the property `favourite` of a contact to turn that contact into a favourite contact. To stress this fact, we choose to assign it to `null`. Conversely, 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, it suffices to check whether `favourite` belongs to its list of properties, which can be done using the program construct `in`.
- **Methods of the Contact Manager.** The method `createContact` creates a new contact and returns it. Given a contact object and an integer n, the method `storeContact` stores the contact corresponding to its first argument in the contact list of the contact manager. As mentioned above, a contact list is an object whose entries are the last names of the stored contacts extended with unique integers to avoid collisions. Hence, the method `storeContact` first checks whether there already exists a contact with the same last name associated with n in the contact list. If it is not the case, it stores the contact in the corresponding property of the contact list. If it is the case, the method calls itself recursively with the same contact but with n incremented by one. Finally, the method `getContact` returns the contact associated with the name and integer given as inputs. If no such contact exists, it returns `undefined`.

Figure 7 presents a typing environment for the Contact Manager example. We diverge from real JavaScript in that we do not specify the type of the prototype of objects of type $\dot{\tau}_{proto\_contact}$. This is equivalent to stating that objects of type $\dot{\tau}_{proto\_contact}$ do not have a prototype, which is not true: in real JavaScript every object has an implicit prototype – `Object.prototype`. This, however, does not compromise the security guarantees offered by the analysis, because not stating the type of the prototype makes the analysis accept less programs than when it is stated but not more. Functions that do not modify the memory are associated with function types with *high* writing effects, which is the most permissive writing effect.

C Proofs

▶ **Lemma 17** (Confined One-Step Transition). *Given a JavaScript configuration* $\langle H, L, E[\bar{s}]\rangle$ *and a monitor configuration* $\langle o, \rho\rangle$ *and a security level* σ*, such that:*

$$\Gamma, pc, \dot{\tau}_r \vdash (\!|\langle H, L, E[\bar{s}]\rangle, \langle o, \rho\rangle|\!) \to (\!|\langle H', L', E'[\bar{s}']\rangle, \langle o', \rho'\rangle|\!)$$

and $\mathsf{lev}(o) \not\sqsubseteq \sigma$*, it holds that:* $\Gamma, pc, \dot{\tau}_r \vdash \langle H, L, E[\bar{s}]\rangle, o, \rho \sim_\sigma \langle H', L', E'[\bar{s}']\rangle, o', \rho'$.

Proof. By case analysis on the structure of $\bar{s}$. ◀

▶ **Lemma 18** (Low-Value Generating One-Step Transitions). *Given two monitored JavaScript configurations and a security level* σ*, such that:*

- $\Gamma, pc, \dot{\tau}_r \vdash (\!|\langle H, L, E[\bar{s}]\rangle, \langle o, \rho\rangle|\!) \to (\!|\langle H_f, L_f, E[\underline{v}]\rangle, \langle o_f, \rho_f\rangle|\!)$ *(hyp.1)*
- $\Gamma, pc, \dot{\tau}_r \vdash (\!|\langle H', L', E[\bar{s}']\rangle, \langle o', \rho'\rangle|\!) \to (\!|\langle H'_f, L'_f, E[\underline{v}']\rangle, \langle o'_f, \rho'_f\rangle|\!)$ *(hyp.2)*
- $\Gamma, pc, \dot{\tau}_r \vdash \langle H, L, E[\bar{s}]\rangle, o, \rho \sim_\sigma \langle H', L', E[\bar{s}']\rangle, o', \rho'$ *(hyp.3)*
- $\mathsf{lev}(o) \sqcup \mathsf{lev}(o') \sqsubseteq \sigma$ *(hyp.4)*

Then, it holds that:

- $H_f \sim_\sigma H'_f$
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$
- $\underline{v}, \mathsf{lev}(\mathsf{head}(\rho_f)) \sim_\sigma \underline{v}', \mathsf{lev}(\mathsf{head}(\rho'_f))$

Proof. We proceed by case analysis in the shape of $\bar{s}$.

[This] Suppose $\bar{s} = \mathsf{this}$ (hyp.5). We conclude that:

- $\underline{v} = H(\mathsf{head}(o), @this)$ and $\underline{v}' = H'(\mathsf{head}(o'), @this)$ — (1) - (hyp.2) + (hyp.3) - (hyp.5)
- $\mathsf{lev}(\Gamma(\mathsf{this})) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ — (2) - (hyp.3) + (1)
- $\mathsf{head}(\rho) = \mathsf{head}(\rho') = \Gamma(\mathsf{this})$ — (3) - (hyp.1)
- $\underline{v}, \mathsf{lev}(\mathsf{head}(\rho)) \sim_\sigma \underline{v}', \mathsf{lev}(\mathsf{head}(\rho'))$ — **(4)** - (2) + (3)
- $H_f = H$, $H'_f = H'$, $L_f = L$, and $L'_f = L'$. — (5) - (hyp.1) + (hyp.2) + (hyp.5)
- $H_f \sim_\sigma H'_f$ — **(6)** - (hyp.3) + (5)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ — **(7)** - (hyp.3) + (5)

[Variable] Suppose $\bar{s} = x$, for some variable x (hyp.5). We conclude that there are two locations l_x and l'_x such that:

- $\bar{s}' = x$ — (hyp.3) - (hyp.5)
- $\underline{v} = H(l_x, x)$ and $\sigma(H, L, x) = l_x$ — (2) - (hyp.1) + (hyp.5)
- $\underline{v}' = H'(l'_x, x)$ and $\sigma(H', L, x) = l'_x$ — (3) - (hyp.2) + (hyp.5)
- $\mathsf{lev}(\Gamma(x)) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ — (4) - (hyp.3) + (2) + (3)
- $\mathsf{head}(\rho_f) = \mathsf{head}(\rho'_f) = \Gamma(x)$ — (5) - (hyp.1) + (hyp.2) + (hyp.5) + (1)
- $\underline{v}, \mathsf{lev}(\mathsf{head}(\rho_f)) \sim_\sigma \underline{v}', \mathsf{lev}(\mathsf{head}(\rho'_f))$ — **(6)** - (4) + (5)
- $H_f = H$, $H' = H'_f$, $L_f = L$, and $L'_f = L'$. — (7) - (hyp.1) + (hyp.2) + (hyp.5)
- $H_f \sim_\sigma H'_f$ — **(8)** - (hyp.3) + (7)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ — **(9)** - (hyp.3) + (7)

[Variable Assignment] Suppose $\bar{s} = x = \underline{v}_0$ for some variable x and value $\underline{v}_0$ (hyp.5). We conclude that:

- $\bar{s}' = x = \underline{v}'_0$ and $\underline{v}_0, \mathsf{lev}(\mathsf{head}(\rho)) \sim_\sigma \underline{v}'_0, \mathsf{lev}(\mathsf{head}(\rho'))$ — (1) - (hyp.3) - (hyp.5)
- $\mathsf{head}(\rho)^{pc} \preceq \Gamma(x)$ and $\mathsf{head}(\rho')^{pc} \preceq \Gamma(x)$ — (2) - (hyp.1) + (hyp.2) + (hyp.5)

- $\underline{v}_0, \mathsf{lev}(\Gamma(x)) \sim_\sigma \underline{v}'_0, \mathsf{lev}(\Gamma(x))$ (3) - (1) + (2)
- $\underline{v} = \underline{v}_0$ and $\underline{v}' = \underline{v}'_0$ (4) - (hyp.1) + (hyp.2) + (hyp.5) + (1)
- $\underline{v}, \mathsf{lev}(\Gamma(x)) \sim_\sigma \underline{v}', \mathsf{lev}(\Gamma(x))$ **(5)** - (3) + (4)
- $H_f \sim_\sigma H'_f$ and $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ **(6)** - (hyp.1) + (hyp.2) + (hyp.5) + (1) + (5)

[Object Literal] Suppose $\bar{s} = \{\,\}\,[\dot{\tau}]$ (hyp.5) for some security type $\dot{\tau}$. We conclude that:

- $\bar{s}' = \{\,\}\,[\dot{\tau}]$ (1) - (hyp.3) - (hyp.5)
- $H_f = H \uplus (l_0, \_proto\_) \mapsto l_{op}$ and $L_f = L$, where: $l_0 = \mathsf{fresh}(H, \dot{\tau})$ (2) - (hyp.1) + (hyp.5)
- $H'_f = H' \uplus (l'_0, \_proto\_) \mapsto l_{op}$ and $L'_f = L'$, where: $l'_0 = \mathsf{fresh}(H', \dot{\tau})$ (3) - (hyp.2) + (1)
- $\Gamma, r \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ **(4)** - (hyp.5) + (2) + (3)

We consider two cases: either the program does a visible object allocation ($\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma$) or the program does an invisible object allocation ($\mathsf{lev}(\dot{\tau}) \not\sqsubseteq \sigma$). Suppose $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma$ (hyp.6):

- $l_0 = l'_0$ (5) - (hyp.1)-(hyp.6)
- $H_f{\restriction^\sigma} = H_f{\restriction^\sigma} \cup \{(l_0, \_proto\_, \mathsf{null}), (l_0, \_proto\_)\}$ (6) - (hyp.6) + (2)
- $H'_f{\restriction^\sigma} = H'_f{\restriction^\sigma} \cup \{(l_0, \_proto\_, \mathsf{null}), (l_0, \_proto\_)\}$ (7) - (hyp.6) + (3) + (5)
- $H_f \sim_\sigma H'_f$ **(8)** - (hyp.3) + (6) + (7)
- $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ **(9)** - (2) + (3) + (5)

Suppose $\mathsf{lev}(\dot{\tau}) \not\sqsubseteq \sigma$ (hyp.6):

- $H_f{\restriction^\sigma} = H{\restriction^\sigma}$ (10) - (hyp.6) + (1)
- $H'_f{\restriction^\sigma} = H'{\restriction^\sigma}$ (11) - (hyp.6) + (2)
- $H_f \sim_\sigma H'_f$ **(12)** - (hyp.3) + (10) + (11)
- $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ **(13)** - (hyp.6)

[Field Projection] Suppose $\bar{s} = l[f]$ (hyp.5). It follows that:

- $\bar{s}' = l'[f']$ and $\langle f, l\rangle, \mathsf{lev}(\lceil\rho\rceil_2) \sim_\sigma \langle f', l'\rangle, \mathsf{lev}(\mathsf{lev}(\lceil\rho'\rceil_2))$ (1) - (hyp.3) - (hyp.5)
- $H_f = H$, $H' = H'_f$, $L_f = L$, and $L'_f = L'$. (2) - (hyp.1) + (hyp.2) + (hyp.5)
- $H_f \sim_\sigma H'_f$ **(3)** - (hyp.3) + (2)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ **(4)** - (hyp.3) + (2)
- $\underline{v} = H(l_f, f)$ and $\underline{v}' = H'(l'_f, f')$, where: $l_f = \pi(H, l, f)$ and $l'_f = \pi(H', l', f')$ (5) - (hyp.1) + (hyp.2) + (hyp.5)

It remains to prove that $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ for $\dot{\tau} = \mathsf{head}(\rho_f)$. Let $\dot{\tau}_o$, $\dot{\tau}_f$, $\dot{\tau}'_o$, $\dot{\tau}'_f$ be the last two elements of ρ and ρ', respectively, and assume that $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma$ (hyp.6); it then follows that:

- $\mathsf{lev}(\dot{\tau}_o) \sqcup \mathsf{lev}(\dot{\tau}_f) \sqsubseteq \sigma$ (6) - (hyp.1) + (hyp.5) + (hyp.6) + (6)
- $\dot{\tau}_o = \dot{\tau}'_o$, $\dot{\tau}_f = \dot{\tau}'_f$, $l_f = l'_f$, and $f = f'$ (7) - (hyp.6) + (1) + (6)
- $\dot{\tau} = \mathsf{head}(\rho_f) = \mathsf{head}(\rho'_f) = \pi(\dot{\tau}_o, f)^{pc \sqcup \mathsf{lev}(\dot{\tau}_o) \sqcup \mathsf{lev}(\dot{\tau}_f)}$ (8) - (hyp.1) + (hyp.2) + (hyp.5) + (1) + (7)
- $\pi(H, l_f, f) = \pi(H', l_f, f)$ (9) - (hyp.3) + (hyp.6) + (7) + (8) + Prototype-Chain Indistinguishability

We consider two cases: $\pi(H, l_f, f) \neq \mathsf{null}$ or $\pi(H, l_f, f) = \mathsf{null}$. Suppose $\pi(H, l_f, f) \neq \mathsf{null}$ (hyp.7), it follows that:

- $\pi(H, l_f, f) = \pi(H', l_f, f) = \hat{l}_{\dot{\tau}'}$ (10) - (hyp.7) + (9)
- $\underline{v} = H(\hat{l}_{\dot{\tau}'}, f)$ and $\underline{v}' = H'(\hat{l}_{\dot{\tau}'}, f)$ (11) - (hyp.1) + (hyp.2) + (hyp.5) + (1) + (10)
- $\dot{\tau}' \preceq \dot{\tau}_o$ (12) - (10) + (11) + Well Labeled Prototype Chains
- $\mathsf{lev}(\dot{\tau}') \sqsubseteq \mathsf{lev}(\dot{\tau}_o) \sqsubseteq \sigma$ (13) - (12) + (hyp.6)
- $\underline{v} = \underline{v}'$ **(14)** - (hyp.3) + (11) + (13)

Suppose $\pi(H, l_f, f) = \mathsf{null}$ (hyp.7):

- $\pi(H', l_f, f) = \mathsf{null}$ (15) - (hyp.7) + (9)
- $\underline{v} = \underline{v}' = \mathsf{undefined}$ **(16)** - (hyp.1) + (hyp.2) + (hyp.7) + (15)

[Membership Check] Suppose $\bar{s} = [f]\,\mathsf{in}\,l$ (hyp.5). It follows that:

- $\bar{s}' = [f']\,\mathsf{in}\,l'$ and $\langle f, l\rangle, \mathsf{lev}(\lceil\rho\rceil_2) \sim_\sigma \langle f', l'\rangle, \mathsf{lev}(\mathsf{lev}(\lceil\rho'\rceil_2))$ (1) - (hyp.3) - (hyp.5)
- $H_f = H$, $H' = H'_f$, $L_f = L$, and $L'_f = L'$. (2) - (hyp.1) + (hyp.2) + (hyp.5)
- $H_f \sim_\sigma H'_f$ **(3)** - (hyp.3) + (2)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ **(4)** - (hyp.3) + (2)

It remains to prove that $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma \Rightarrow \underline{v} = \underline{v}'$ for $\dot{\tau} = \mathsf{head}(\rho_f)$. Let $\dot{\tau}_o$, $\dot{\tau}_f$, $\dot{\tau}'_o$, $\dot{\tau}'_f$ be the last two elements of ρ and ρ', respectively, and assume that $\mathsf{lev}(\dot{\tau}) \sqsubseteq \sigma$ (hyp.6); it then follows that:

- $\dot{\tau} = \mathsf{PRIM}^{\bar{\pi}(\dot{\tau}_o, f)\sqcup\mathsf{lev}(\dot{\tau}_o)\sqcup\mathsf{lev}(\dot{\tau}_f)}$ (5) - (hyp.1) + (hyp.5)
- $\dot{\tau}_o = \dot{\tau}'_o$, $\dot{\tau}_f = \dot{\tau}'_f$, $l = l'$, and $f = f'$ (6) - (hyp.6) + (1) + (5)
- $\dot{\tau} = \mathsf{head}(\rho_f) = \mathsf{head}(\rho'_f) = \mathsf{PRIM}^{\bar{\pi}(\dot{\tau}_o, f)\sqcup\mathsf{lev}(\dot{\tau}_o)\sqcup\mathsf{lev}(\dot{\tau}_f)}$ (7) - (hyp.1) + (hyp.2) + (hyp.5) + (1) + (6)
- $\pi(H, l_f, f) = \pi(H', l_f, f)$ (8) - (hyp.3) + (hyp.6) + (6) + (7) + Prototype-Chain Indistinguishability

We consider two cases: $\pi(H, l_f, f) \neq \mathsf{null}$ or $\pi(H, l_f, f) = \mathsf{null}$. Suppose $\pi(H, l_f, f) \neq \mathsf{null}$ (hyp.7), it follows that:

- $\pi(H, l_f, f) = \pi(H', l_f, f) = \hat{l}_{\dot{\tau}'}$ (9) - (hyp.7) + (8)
- $\underline{v} = \mathsf{true}$ and $\underline{v}' = \mathsf{true}'$ **(10)** - (hyp.1) + (hyp.2) + (hyp.5) + (1) + (9)

Suppose $\pi(H, l_f, f) = \mathsf{null}$ (hyp.7):

- $\pi(H', l_f, f) = \mathsf{null}$ (11) - (hyp.7) + (8)
- $\underline{v} = \underline{v}' = \mathsf{undefined}$ **(12)** - (hyp.1) + (hyp.2) + (hyp.7) + (11)

[Field Assignment] Suppose $\bar{s} = l[f] = \underline{v}$ (hyp.5). It follows that:

- $\bar{s}' = l'[f'] = \underline{v}'_0$ and $\langle \underline{v}, f, l\rangle, \mathsf{lev}(\lceil\rho\rceil_3) \sim_\sigma \langle \underline{v}', f', l'\rangle, \mathsf{lev}(\mathsf{lev}(\lceil\rho'\rceil_3))$ (1) - (hyp.3) - (hyp.5)
- $H_f = H[l.f \mapsto \underline{v}]$, $L_f = L$, $H'_f = H'[l'.f' \mapsto \underline{v}']$, and $L'_f = L'$ (2) - (hyp.1) + (hyp.2) + (hyp.5)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ and $\underline{v}, \mathsf{head}(\rho_f) \sim_\sigma \underline{v}', \mathsf{head}(\rho'_f)$ **(3)** - (hyp.3) + (2)

Let $\dot{\tau}_v$, $\dot{\tau}_f$, $\dot{\tau}_o$, $\dot{\tau}'_v$, $\dot{\tau}'_f$, and $\dot{\tau}'_o$ be the last three elements of ρ and ρ' respectively. Suppose that $pc \sqcup \mathsf{lev}(\dot{\tau}_o) \sqcup \mathsf{lev}(\dot{\tau}_f) \sqsubseteq \sigma$ for $\dot{\tau} = \mathsf{head}(\rho_f)$ (hyp.6), it follows that:

- $\dot{\tau}_o = \dot{\tau}'_o$ and $\dot{\tau}_f = \dot{\tau}'_f$, $l = l'$, and $f = f'$ **(4)** - (hyp.6) + (1)
- $\dot{\tau}_v^{pc\sqcup\mathsf{lev}(\dot{\tau}_o)\sqcup\mathsf{lev}(\dot{\tau}_f)} \preceq \dot{\tau}_o.f$ and $(\dot{\tau}'_v)^{pc\sqcup\mathsf{lev}(\dot{\tau}_o)\sqcup\mathsf{lev}(\dot{\tau}_f)} \preceq \dot{\tau}_o.f$ (5) - (hyp.1) + (hyp.2) + (4)
- $\underline{v}, \dot{\tau}_o.f \sim_\sigma \underline{v}', \dot{\tau}_o.f$ (6) - (1) + (5)
- $H_f \sim_\sigma H'_f$ **(7)** - (hyp.3) + (3) + (4)

Suppose that $pc \sqcup \mathsf{lev}(\dot{\tau}_o) \sqcup \mathsf{lev}(\dot{\tau}_f) \not\sqsubseteq \sigma$ for $\dot{\tau} = \mathsf{head}(\rho_f)$ (hyp.6), it follows that:

- $\dot{\tau}_o.\overline{f} \sqcap \dot{\tau}_o.\overline{f} \not\sqsubseteq \sigma$ (8) - (hyp.1) + (hyp.2) + (hyp.6)
- $H_f \sim_\sigma H'_f$ **(9)** - (hyp.3) + (8)

[Field Deletion] Suppose $\bar{s} = \mathsf{delete}\,l\,f$ (hyp.5). It follows that:

- $\bar{s}' = \mathsf{delete}\,l'\,f'$ and $\langle f, l\rangle, \mathsf{lev}(\lceil\rho\rceil_2) \sim_\sigma \langle f', l'\rangle, \mathsf{lev}(\mathsf{lev}(\lceil\rho'\rceil_2))$ (1) - (hyp.3) - (hyp.5)
- $H_f = H\backslash l.f$, $L_f = L$, $H'_f = H'\backslash l'.f'$, and $L'_f = L'$ (2) - (hyp.1) + (hyp.2) + (hyp.5)
- $\Gamma \vdash H_f, L_f \sim_\sigma H'_f, L'_f$ **(3)** - (hyp.3) + (2)

Let $\dot{\tau}_f$, $\dot{\tau}_o$, $\dot{\tau}'_f$, and $\dot{\tau}'_o$ be the last two elements of ρ and ρ' respectively. Suppose that $pc \sqcup \mathsf{lev}(\dot{\tau}_o) \sqcup \mathsf{lev}(\dot{\tau}_f) \sqsubseteq \sigma$ for $\dot{\tau} = \mathsf{head}(\rho_f)$ (hyp.6), it follows that:

- $\dot{\tau}_o = \dot{\tau}'_o$ and $\dot{\tau}_f = \dot{\tau}'_f$, $l = l'$, and $f = f'$ (4) - (hyp.6) + (1)

- $\mathsf{lev}(\dot\tau_o) \sqcup \mathsf{lev}(\dot\tau_f) \sqsubseteq \dot\tau_o.\overline{f}$ and $\mathsf{lev}(\dot\tau_o) \sqcup \mathsf{lev}(\dot\tau_f) \sqsubseteq \dot\tau_o.\overline{f}$ (5) - (hyp.1) + (hyp.2) + (4)
- $H_f \sim_\sigma H'_f$ **(6)** - (hyp.3) + (4) + (5)

Suppose that $pc \sqcup \mathsf{lev}(\dot\tau_o) \sqcup \mathsf{lev}(\dot\tau_f) \not\sqsubseteq \sigma$ for $\dot\tau = \mathsf{head}(\rho_f)$ (hyp.6), it follows that:

- $\dot\tau_o.\overline{f} \sqcap \dot\tau_o.\overline{f} \not\sqsubseteq \sigma$ (7) - (hyp.1) + (hyp.2) + (hyp.6)
- $H_f \sim_\sigma H'_f$ **(8)** - (hyp.3) + (8)

The remaining cases follow similarly. ◀

▶ **Lemma 19** (Monitored Execution of Typable Redex). *Given an expression redex $\bar e$, a typing environment Γ, a security level pc, a security type $\dot\tau$, a heap H, a scope chain L, a control context o, and an expression context ρ such that:*

- $\Gamma, pc \vdash_e \bar e : \dot\tau$ *(hyp.1)*
- $\langle H, L, \bar e\rangle \xrightarrow{\alpha_l} \langle H', L, v\rangle$ *(hyp.2)*

Then, it holds that:

$$\Gamma, pc, \dot\tau_r \vdash (\!|\langle H, L, \bar e\rangle, \langle o, \rho\rangle|\!) \to (\!|\langle H', L, v\rangle, \langle o, \dot\tau' :: \rho'\rangle|\!)$$

where: $\rho' \sqsubseteq \rho$ and $\dot\tau' = \dot\tau$.

Proof. We proceed by case analysis on $\bar e$.

[This] Suppose $\bar s = \mathsf{this}$ (hyp.3). We conclude that:

- $\rho' = \rho$ and $\dot\tau' = \Gamma(this)$ (1) - (hyp.2) + (hyp.3)
- $\dot\tau = \Gamma(this)$ (3) - (hyp.1) + (hyp.3)
- $\dot\tau = \dot\tau'$ **(4)** - (2) + (3)

[Variable] Suppose $\bar s = x$ (hyp.3). We conclude that:

- $\rho' = \rho$ and $\dot\tau' = \Gamma(x)$ (1) - (hyp.2) + (hyp.3)
- $\dot\tau = \Gamma(x)$ (2) - (hyp.1) + (hyp.3)
- $\dot\tau = \dot\tau'$ **(3)** - (2) + (3)

[Object Literal] Suppose $\bar s = \{\,\}\,[\dot\tau'']$ (hyp.3). We conclude that:

- $\rho' = \rho$ and $\dot\tau' = \dot\tau''$ (1) - (hyp.2) + (hyp.3)
- $\dot\tau = \dot\tau''$ (2) - (hyp.1) + (hyp.3)
- $\dot\tau = \dot\tau'$ **(3)** - (2) + (3)

[Project Literal] Suppose $\bar s = l[f]$ (hyp.3). Letting $\rho = \dot\tau_f :: \dot\tau_o :: \rho''$, we conclude that:

- $\rho' = \pi(\dot\tau_o, f)^{pc \sqcup \mathsf{lev}(\dot\tau_o) \sqcup \mathsf{lev}(\dot\tau_f)} :: \rho''$ and $\dot\tau' = \pi(\dot\tau_o, f)^{pc \sqcup \mathsf{lev}(\dot\tau_o) \sqcup \mathsf{lev}(\dot\tau_f)}$ **(1)** - (hyp.2) + (hyp.3)
- $\dot\tau = \dot\tau'$ (2) - (hyp.1) + (hyp.3)

The remaining cases follow by similar arguments. ◀

▶ **Lemma 20** (Monitored Execution of Typable Redex). *Given a statement s, a typing environment Γ, a security level pc, two security types $\dot\tau$ and $\dot\tau_r$, a heap H, a scope chain L, a control context o, and an expression context ρ such that:*

- $\Gamma, pc, \dot\tau_r \vdash_s s \hookrightarrow s'$ *(hyp.1)*
- $\langle H, L, s\rangle \to^* \langle H', L, v\rangle$ *(hyp.2)*

Then, it holds that:

$$\Gamma, pc, \dot\tau_r \vdash (\!|\langle H, L, s\rangle, \langle o, \rho\rangle|\!) \to^* (\!|\langle H', L, v\rangle, \langle o, \rho\rangle|\!)$$

Proof. By induction on the length of the derivation of (hyp.2) using the previous lemma. ◀

▶ **Theorem 21** (Noninterference)**.** *For any typing environment* Γ*, levels* σ *and pc, security type* $\dot{\tau}$*, statement,* s*, and two heaps* H_0 *and* H_1*, such that* $\Gamma, pc, \dot{\tau} \vdash_s s \hookrightarrow s'$*,* $H_0 \sim_\sigma H_1$*, and* $(\!|\langle H_i, [\,], s'\rangle, \{\}|\!) \to^* (\!|\langle H_i', [\,], v_i\rangle, \{\}|\!)$ *for* $i = 0, 1$*, it holds that* $H_0' \sim_\sigma H_1'$*.*

Proof. By induction on the length of the derivation of $(\!|\langle H_0, [\,], s'\rangle, \{\}|\!) \to^* (\!|\langle H_0', [\,], v_0\rangle, \{\}|\!)$ using Lemmas 20 and 18. ◀