Value-sensitive Hybrid Information Flow Control for a JavaScript-like Language

Daniel Hedin
Mälardalen University
Chalmers University of Technology

Luciano Bello
Chalmers University of Technology

Andrei Sabelfeld
Chalmers University of Technology

***Abstract*—Secure integration of third-party code is one of the prime challenges for securing today's web. Recent empirical studies give evidence of pervasive reliance on and excessive trust in third-party JavaScript, with no adequate security mechanism to limit the trust or the extent of its abuse. Information flow control is a promising approach for controlling the behavior of third-party code and enforcing confidentiality and integrity policies. While much progress has been made on static and dynamic approaches to information flow control, only recently their combinations have received attention. Purely static analysis falls short of addressing dynamic language features such as dynamic objects and dynamic code evaluation, while purely dynamic analysis suffers from inability to predict side effects in non-performed executions. This paper develops a value-sensitive hybrid mechanism for tracking information flow in a JavaScript-like language. The mechanism consists of a dynamic monitor empowered to invoke a static component on the fly. This enables us to achieve a sound yet permissive enforcement. We establish formal soundness results with respect to the security policy of noninterference. In addition, we demonstrate permissiveness by proving that we subsume the precision of purely static analysis and by presenting a collection of common programming patterns that indicate that our mechanism has potential to provide more permissiveness than dynamic mechanisms in practice.**

I. Introduction

Web applications are frequently built using code from different sources. The script inclusion mechanism provides a simple integration platform for loading third-party scripts (usually written in JavaScript) into users' browsers. This approach is powerful because the code operates with full access to the users' credentials and with the same privileges as the page that includes it.

Motivation: Secure code integration. Unfortunately, this power opens up for abusing the trust, either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. Indeed, a recent empirical study [42] of script inclusion reveals excessive reliance on and excessive trust in third-party scripts. It confirms that a vast majority of today's web pages (including sensitive services as banks and online shopping [29], [42]) include third-party scripts. Regretfully, the current security practice falls short of distinguishing a benign analytics script from a malicious script that leaks users' information to an attacker. In typical third-party code scenarios, such as usage analytics, advertisement, helper libraries, traditional access control (as supported at Internet domain level by the SOP [43], CSP [56], CORS [57] policies) is of limited help. The code must be granted access and execution rights for proper functionality. Of paramount importance is what the code does with the data after permission has been granted.

Need for practical information flow control. To address this, tracking *information flow* throughout the program execution is a promising technique for preventing sensitive information from being divulged to unauthorized parties.

Static vs. dynamic information flow control. Much progress has been made on information flow control for more and more expressive languages. The approaches range from purely *static* [8], [16], [31], [40], [46], [50], [55] to purely *dynamic* [3], [17], [21], [24], [27], [29], [47], [51], [54] and increasingly popular *hybrid* [35], [37], [44], [49], [53].

Unfortunately, *static* analysis falls short of addressing the highly dynamic features of JavaScript [29], [52]. The obvious roadblocks include dynamic code evaluation and the possibility to dynamically modify the structure of objects together with aliasing.

An arguably better fit for information-flow control enforcement for JavaScript is *dynamic* information-flow analyses. A dynamic information-flow analysis works essentially as a dynamic type system: at runtime each value is tagged with a security label that represents the security classification of that value. The security labels are then updated during each computation to model the information flow, e.g., the label of the result of an addition is the join of the labels of the addends.

The key benefit for dynamic analysis over static analyses is the availability of runtime values. However, sound purely dynamic information-flow analyses come with an inherent demand that places fundamental limitations on pure dynamic enforcement: for the analysis to be sound, it is important that the security labels themselves are not dependent on secrets [4], [44].

Motivated by the above considerations, this paper presents a sound hybrid monitor based on a dynamic information flow analysis that makes use of a static component at key points during the execution in order to alleviate the limitations of pure dynamic enforcement. A key challenge is to design a combination that benefits from the respective advantages rather than suffering from the respective disadvantages. We now briefly discuss the considerations for such a design.

Explicit vs. implicit flows. There are two basic categories of information flow: *explicit* and *implicit* flow. Explicit flows come from explicit actions, like storing information in a variable, or sending information over the network. Implicit flows, on the other hand, are caused by the control flow, like the following example:

```
var l = false; if (h) {l = true;}
```
Example 1

Although there is no explicit flow from `h` to `l`, the program has the effect of leaking information about `h` into `l`.

Permissiveness of monitors. A common way to enforce independence of labels is collectively known as *no-sensitive-upgrade (NSU)* [4], [58]. NSU achieves independence by preventing labels from changing under *secret control* (or *secret context*), i.e., when computation finds itself in a control-flow region whose reachability depends on secrets. When label upgrade under secret control is attempted, NSU blocks the execution. Thus, a purely dynamic monitor will cause the execution of Example 1 to stop in case `h` is true and secret. Notice that this stopping might be premature, since the `l`, while carrying information from `h`, might never end up being observable by the attacker.

As a consequence, many efforts have been directed to address this limitation to practical dynamic information flow control, driven by the desire to permit code to make changes in labels in secret contexts while the program as a whole remains secure. The goal is to extend *permissiveness* (low number of false positives) while maintaining *soundness* (no false negatives) of dynamic enforcements. This can be achieved by identifying potential *write targets* and upgrading them before entering *elevated contexts*. Annotations like *privatization operations* [5] or *upgrade annotations* [29] have been proposed to allow programmers to pass dynamic monitors additional information. In the above example, such an annotation could take the form of an upgrade instruction that upgrades the variable l before entering the secret conditional. The downside with such annotations is that they have to be added to the program by some means (manually, or via testing [12]) and that the expressive power of the annotations is limited by the annotation language.

Our solution: Hybrid information flow enforcement. This paper sets out for a sweet spot between static and dynamic analysis: a sound modular hybrid monitor based on a dynamic information flow analysis that makes use of a static component just before the execution of elevated contexts in order to identify potential write targets and upgrade their labels. There are three key ideas underlying the hybrid monitor. First, the static component is *value sensitive* in that it makes use of *public values* in the environment for the approximation of write targets. This is fundamental for the treatment of the heap and closures. Second, the static component is only used to improve the *permissiveness* of the overall analysis, while *soundness* is provided by the base-line dynamic monitor. This entails that the static component is able to ignore potential write targets when they cannot be precisely established. Third, the monitor is modular in the sense that it decouples the dynamic and static parts with a clear semantic interface between the two. This allows for future development in order to improve permissiveness and performance, while simplifying the soundness argument. Our monitor generalizes previous work on sound hybrid enforcement for simple imperative languages with variables [35], [37], [44], [49] and contributes to bridging the gap to hybrid security analyses for JavaScript that come without soundness guarantees (e.g. [53]).

Contributions. The main contribution of this paper is a hybrid monitor that emphasizes permissiveness without sacrificing soundness. The hybrid monitor is able to deal with a set of language constructions selected to capture several challenges of JavaScript: 1) dynamic records, existence of properties and structure, 2) dynamic scope chain and `with`, 3) `eval`, 4) closures and `return`. This set constitutes a core of JavaScript, leading us to a path from theory to practice that mirrors the paths taken previously by the related efforts: from theory of dynamic information flow control for JavaScript [29] to practice [27], and from theory of secure multi-execution [17] to practice for JavaScript [24].

Further, we demonstrate the advance of permissiveness with respect to the state of the art in three ways. First, we show that the permissiveness of our hybrid mechanisms subsumes a static flow-sensitive analysis in the style of a classical analysis by Hunt and Sands [31], generalized to treat the heap. Second, we demonstrate that we gain permissiveness over purely dynamic analysis [29] on a collection of programming patterns that we have observed in empirical JavaScript studies with programs found on actual web pages. Third, we show that our hybrid approach enjoys the same benefit over static approaches as the approach taken in testing-based program analysis [12] without the extra complication of specialized upgrade instructions and reliance on program path coverage by the testing.

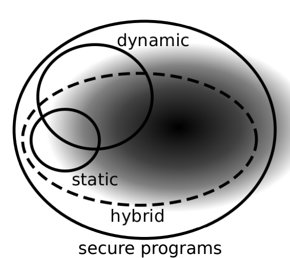


Fig. 1: Relative permissiveness

The relation between the approaches is illustrated in Figure 1, where the gray area represents the programs found in the wild. From a practical standpoint, performing well on this set of programs is important and, as shown in the figure, our experience indicate that our approach improves over purely dynamic monitors on this set. Experimentally verifying the exact extent of the gray area requires scaling the approach to full JavaScript. This work is a step in that direction.

Further, the inclusion of the set labeled *static* into the set labeled *hybrid* depicts that the hybrid mechanism subsumes the static, i.e., that all programs accepted by the static analysis are also accepted by our hybrid approach. However, no formal subsumption exists between the purely dynamic analysis and the other approaches, elaborated in Section VI.

Finally, we have implemented our monitor in Haskell and used the implementation to evaluate the approach. The source code is available at [1] together with an online interpreter. The interpreter provides all examples of this paper for easy execution, but we encourage the reader to experiment. The online interpreter is explained in the full version of this paper, also available at [1].

II. CHALLENGES

JavaScript is a highly dynamic language with a number of features that pose challenges for both dynamic and hybrid enforcements. The goal of these enforcements is to avoid premature stopping, while allowing these features and maintain soundness.

Dynamic objects. JavaScript features *dynamic objects* in the sense that properties can be added and deleted at runtime. This possibility leads to the situation that the presence and absence of properties can be used to encode secrets. One way of dealing with this is by associating security labels with the *existence* of properties and the *structure* of objects [29]. The latter is used to record the security label associated with the absence of properties.

Consider Example 2, where a property is added under secret control. After execution, the presence of `x` in `o` implies that `h` is `true` and its absence that `h` is `false`. Since the structure is initially public, this means that the structure security label would have to be changed under secret control, which cannot be allowed. Thus, for unhindered execution, the structure security label must be upgraded before executing the secret conditional.

```
var o = {}; if (h) {o["x"] = true;}
```
Example 2

Compare with Example 3, where the existing property `x` is updated in a secret conditional. This implies that the security label of the value of `x` would be changed under secret control, which requires that it is upgraded before the secret conditional.

```
var o = {x : false};
if (h) {o["x"] = true;}
```
Example 3

Dynamic scope chain. Another challenge with JavaScript is the dynamic scope chain: variables can be dynamically declared (via, e.g., `eval`) and dynamic objects can be injected into the scope chain using `with`. Variable lookup is performed by traversing the scope chain, searching each environment record until a binding of the variable is found.

In Example 4 assigning to `x` updates the variable, whereas assigning to `y` writes to a property of `o`. This implies that the security labels of the value of `x` and of the value of property `y` would be changed under secret control, which requires that they are upgraded before the secret conditional.

```
var x; var o = {y : false};
with (o) {if (h) {x = true; y = true;}}
```
Example 4

In addition, this illustrates that the structure of the environment records on the scope chain affects the reads and the writes. Consider Example 5, where the presence of field in `o` is secret. Since `o` is injected into the scope chain, this information is encoded by the variable update: assigning to `x` will write to the topmost variable `x`, encoding the fact that no property `x` was present in `o`. Similarly, assigning to `y` will not write to the topmost variable `y` but rather the property `y` of `o`, encoding that `o` has property `y`. This requires that the security labels on variables `x` and (perhaps somewhat surprising) `y` are upgraded before performing the actual assignments.

```
var x; var y; var o = {  };
if (h) {o["y"] = true;};
with (o) {x = true; y = true;}
```
Example 5

Return. For most control flow constructs the secret context follows the syntactic structure of the program, i.e., the extent of the secret context initiated by a secret conditional is the body of the conditional. However, the possibility to `return` from within secret contexts breaks this property. Consider Example 6 where the value of `h` controls if the assignment `x=1` is executed or not. One way of viewing this is that a return in a secret context has the effect of extending the secret context to the end of the function. In the example, this implies that the security label of the value of `x` would be changed under secret control, which requires that it is upgraded before the secret conditional.

```
var x;
(function() {
  if (h) {return true;} x = 1; return false;
})();
```
Example 6

Closures. Since JavaScript does not contain any constructs for information hiding like *protected* or *private* properties it is common to use function closures to mimic this behavior.

For instance, consider Example 7, which implements a one-place memory. This is achieved by letting the state (the variable `data`) be declared in a function, and returning an object that provides functions to interact with the state. In this case there are two functions `set` and `get` that sets and gets the value respectively. In reality, this pattern would be used for more interesting data structures, e.g. stacks or hash tables, but for illustration a one place memory suffices.

Now, imagine the situation in the example. The data structure is allocated at the start of the program, outside any secret contexts. This means that the label of `data` will be public. Thereafter, the program interacts with the data structure by calling `set` from within the secret context. This requires the security label of the value of `data` to be upgraded before the secret conditional.

```
var mem = function() {
  var data = null;
  return {set : function(d) {data = d;},
          get : function() {return data;} };
};
var x = mem(); if (h) {x["set"](true);}
```
Example 7

The example is abstracted from code found in Google Analytics [23]; in our experience this kind of situation is frequently occurring in production code.

Dynamic code execution. JavaScript provides several ways of performing dynamic code execution with `eval` being the most prominent. Given a string, `eval` parses it as a program and executes the result. The runtime aspect of eval poses no significant challenge for dynamic or hybrid analysis. However, `eval` allows for runtime declaration of variables, which causes information flow challenges similar to `with` as discussed above.

Consider the assignment to `l` in Example 8. If `h` is `true` the local scope of `f` will contain `l` and the update is local. On the other hand, if `h` is `false` the outer variable `l` will be updated. This implies that the security label of the value of the outer `l` must be upgraded before the assignment.

```
var l = true;
(function() {
  if (h) {eval("var l;");} l = false;
})();
```
Example 8

III. Language

In this paper we target a carefully selected set of language constructs that illustrate the core hybrid information-flow principles needed to analyze full JavaScript. The language we propose is a small imperative language with `eval`, dynamic records, first class functions and a variable scope chain that terminates in a global record and allows for the injection of user defined records into the scope chain using `with`. In addition, like JavaScript, we employ non-syntactic scoping, variable and function hoisting and the principle that writing to previously undeclared variables defines them in the outermost scope, i.e., the global record.

The dynamic records represent the objects of JavaScript and capture the key challenge from an information flow perspective: that properties can be added and deleted from records (objects) at runtime.

$$
\begin{array}{rcl}
e & ::= & x \mid l \mid e \oplus e \mid e[e] \mid x := e \mid e[e] := e \\
& \mid & \texttt{function}\ (\overline{x})\ s \mid e(\overline{e}) \mid \{\overline{x} : \overline{e}\} \\
s & ::= & \texttt{var}\ x \mid s ; s \mid \texttt{if}\ e\ s\ s \mid \texttt{while}\ e\ s \mid \texttt{return}\ e \\
& \mid & \texttt{with}\ e\ s \mid \texttt{eval}\ e \mid e
\end{array}
$$

Fig. 2: Syntax

For clarity of exposition we make a number of simplifications. On the more mundane side, we collapse the primitive values of JavaScript to a single category of abstract literals, represent all operators by one binary operator, omit the distinguished `undefined` value, among others. More fundamentally, we do not model prototype based inheritance, exceptions, accessor properties (getters and setters), property attributes, or implicit coercions. Additionally, modeling the standard JavaScript API, while necessary for a full scale implementation, is out of scope of this paper. See Section IV-D for a more detailed discussion on scaling to the full language.

The syntax of the language (Figure 2) is built up by two main syntactic categories: the expressions e, and the statements s. The expressions consist of variables, primitive literals, binary operators, property projection, variable and projection update, function expressions, function call and object literals. The statements consist of variable declaration, sequencing, conditional branching and iteration, a `return` statement that stops the execution of a function returning the given value, the `with` statement that takes a record and a statement and injects it into the scope chain before executing the statement, and the `eval` statement that takes a string, parses it and executes the result. Finally, expressions are lifted into the statements.

The semantics for this subset of JavaScript is standard. We refer the reader to [29] for a more detailed operational explanation and purely dynamic information-flow semantics.

IV. Enforcement

As illustrated before, purely dynamic enforcement of information-flow often fails to handle common programming patterns found in web applications. The problem is that a purely dynamic analysis is limited to make security decisions based on a single trace while enforcing a property on sets of traces. A static analysis does not have this problem, since all possible paths are considered. On the other hand, a static analysis typically has limited information about runtime values making them ill-suited for dynamic languages like JavaScript.

We bridge these limitations by combining the dynamic and static approaches to create a hybrid monitor that inherits the benefits of both. More precisely, we extend a dynamic monitor with a static component that is applied whenever there is an elevation of the security context, e.g., at secret conditionals. The static component analyzes the extent of the secret context, e.g., the body of the conditional, and upgrades potential write targets that otherwise might cause the dynamic analysis to block the execution. After the static component is done, execution proceeds normally under the dynamic monitor. The fact that execution continues monitored, and hence is subject to the NSU restriction, is important. This way the soundness of the hybrid monitor is ensured by the soundness of the dynamic monitor which gives us freedom in the design of the static component. In particular it allows us to ignore cases where we cannot compute the write locations precisely instead of being overly pessimistic.

The rest of this section is laid out as follows. First, Section IV-A introduces the values and the execution environment of the language: primitive values, records, scopes, scope chains, environments and configurations. Thereafter, Section IV-B presents the hybrid monitor by discussing key constructions from the perspective of the limitations of pure dynamic information flow enforcement and how the static component is used to increase permissiveness. This section is written in relation to an intuitive understanding of how the static component upgrades potential write targets. Finally, details on how the static component computes potential write targets are presented in Section IV-C.

$$
\begin{array}{c}
v ::= l \mid p \mid \texttt{null} \mid \langle \overline{x}, s, \gamma \rangle \quad o ::= \langle \phi, \varsigma \rangle \\
\gamma ::= \langle \gamma, p^{\sigma} \rangle \mid \texttt{null} \quad h = p \mapsto o \quad \phi = f \rightharpoonup v^{\sigma} \\
E ::= \langle h, \gamma, \eta \rangle \quad \mathcal{C} ::= E \mid \langle E, v^{\sigma} \rangle
\end{array}
$$

Fig. 3: Values

A. Execution environment

Let x and f range over identifiers. The values of the language (Figure 3) are the literals l, the pointers p, the distinguished `null` pointer as well as closures $\langle \overline{x}, s, \gamma \rangle$ representing function closures. In the following we identify meta variables with the sets that they range over, e.g., v denotes both the set of values as well as the meta variable that ranges over the set of values.

Let σ range over the set of security labels in general and let ς denote the structure security label, η denote the return context label (the highest security context in which return is allowed to execute) and pc denote the program counter label in particular. Without loss of generality the security labels are drawn from a two level security lattice defined by $L \sqsubseteq H$, where L and H denotes public and secret information respectively. The extension to an arbitrary lattice is possible, but demands that all definitions be parametrized over a security level corresponding to the attackers view which clutters the exposition. The security of the general lattice is formulated as the preservation of noninterference for any order preserving mapping of the general lattice onto the two level lattice. Hence, the notion is by definition attacker agnostic; regardless of where the attacker is in the lattice he cannot learn anything above him.

All values occur labeled with security labels. To reduce clutter, we frequently omit writing out the labels explicitly whenever they do not take active part in the computation. This is indicated by a dot over the corresponding meta variable, e.g., $\dot{v}$ denotes a labeled value and $\dot{p}$ denotes a labeled pointer. For such values the notation $\dot{v}^{\sigma}$ denotes joining σ with the hidden label, i.e, $\dot{v}^{\sigma_2} = v^{\sigma_1 \sqcup \sigma_2}$ if $\dot{v} = v^{\sigma_1}$.

A record $\langle \phi, \varsigma \rangle$ is a dynamically modifiable map from property names to labeled values paired with the structure label of the record, ς. The property map, ϕ, is a *labeled partial map*, written $f \rightharpoonup \dot{v}$, where each association, $f \overset{\sigma}{\mapsto} v$, in the map is labeled with an existence security label, σ.

The variable environment is built by a scope chain where each scope $\langle \gamma, p^{\sigma} \rangle$ contains a labeled pointer to a record containing the actual bindings as well as the inner scope. The scope chain is terminated with the global scope, $\langle \texttt{null}, p_g^L \rangle$, where p_g is the distinct pointer to the global record. This allows us to model key features of the JavaScript variable binding like the global object and `with`.

The heap is simply a map from pointers to records and the environments are triples $\langle h, \gamma, \eta \rangle$ consisting of a heap,

a scope chain and the return label. The return label is part of the environment, since it, unlike the *pc*, does not follow the syntactic structure of the program, see Example 6 and Section IV-B for more information.

Finally, configurations $\mathcal{C}$ are either environments, or pairs of environments and values. The latter indicates that a *return* statement was executed and that execution should return to the caller.

B. Hybrid monitor

The hybrid monitor semantics is of the form $\langle E, s\rangle \xrightarrow{pc} \mathcal{C}$, read as the statement s executes in environment E under the program counter, pc, to a configuration $\mathcal{C}$. The pc is the standard way to prevent implicit flows. For control structures like conditional branches the pc is raised to the security label of the guard and remains so for the extent of the control structure. Together with the return label of the environment the pc forms the *security context* that is part of the governing of side effects. In particular, implicit flows are then prevented by forbidding all side effects with targets that are below the security context.

In the setting of runtime monitoring forbidding side effects translates to stopping the execution, typically achieved by not providing rules for execution for such situations. Thus instead of causing an explicit error, the semantics fails to progress.

Figure 4 contains a selection of the semantic rules of the monitor. We refer the reader to the full version of this paper [1] for the remaining rules. The hybrid monitor is based on a standard purely dynamic monitor extended with a static component used in language constructions that can cause (extensions to) elevated *write contexts* (the security context together with labels of values deciding the target of the write, see below). The static component approximates the potential write targets and updates their security labels before execution continues in the dynamic (part of the hybrid) monitor. This way the static component decouples the update from secret control and prevents the execution from being stopped for security reasons.

The hybrid rules that trigger the static component are: conditional branches, sequences (triggered by `return`), variable assignments (internally, due to scope chain traversal), function call and `eval`. Of those, the conditional branch provides the most direct illustration of write context elevation and how the static component is applied. Consider rule IF-H, where the label of the guard, σ, is not below the security context, $pc \sqcup \eta_2$. This means that the executed branch, s_b, will be executed in an elevated security context. For this reason, the rule applies the corresponding static component, i.e., the static semantics for statements (denoted $\Rightarrow$), on both branches in order to upgrade potential write targets before the execution of the selected branch. The remaining rules that cause elevated security contexts work analogously, but the context elevation is more intricate and manifested by interaction of several rules potentially spanning the hybrid monitor and the static component.

Elevated security contexts are not the only source of elevated write contexts. As illustrated by Example 4 and Example 5 the dynamic scope chain gives rise to elevated write contexts for variable update. To handle this, rule VASSIGN applies the static component in the form of static versions of *find*, $\text{find}(\!|\cdot|\!)$, and record update, $\cdot[\![\cdot \leftarrow \cdot]\!]$, to upgrade the potential write target before the actual update is performed.

Below we first discuss how the notion of write context is used in restricting side effects in the language. Thereafter, we explain conditional branches, non-syntactic control flow, variable assignment, function call and eval in greater detail. The static component is described in detail in Section IV-C.

The write context. All side effects of the language are formulated in terms of record update via VASSIGN and PASSIGN. There are two rules for record update: RECUP-1 that updates an existing property and RECUP-2 that add a new property. The rules are parameterized over the security context, ctx, of the update, i.e., the join of the pc and the return label.

$$\frac{f \xmapsto{\sigma_1} w^{\sigma_w} \in \phi_1 \quad \phi_2 = \phi_1[f \xmapsto{\sigma_1 \sqcap \sigma_2} v^{\sigma_2}] \quad \langle \phi_1, \varsigma\rangle = h[p] \quad \sigma_2 = ctx \sqcup \sigma_p \sqcup \sigma_f \quad \sigma_2 \sqsubseteq \sigma_w}{h[(p^{\sigma_p})[f^{\sigma_f}] \xmapsto{ctx} v] = h[p \mapsto \langle \phi_2, \varsigma\rangle]} \text{ RECUP-1}$$

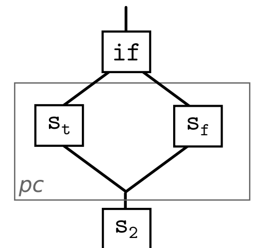
$$\frac{\sigma \sqsubseteq \varsigma \quad \phi_2 = \phi_1[f \xmapsto{\sigma \sqcup \sigma_f} v^{\sigma \sqcup \sigma_f}] \quad \langle \phi_1, \varsigma\rangle = h[p] \quad f \notin dom(\phi_1) \quad \sigma = ctx \sqcup \sigma_p}{h[(p^{\sigma_p})[f^{\sigma_f}] \xmapsto{ctx} v] = h[p \mapsto \langle \phi_2, \varsigma \sqcup \sigma_f\rangle]} \text{ RECUP-2}$$

When writing to a record not only the value of the property is written, but also the structure of the record is affected, i.e., which properties are defined. For this reason there are two different write contexts at play in the rules. First, the write context of the record is the security context together with the security label of the pointer σ_p. Second, the write context of the property is the security context together with the security label of the pointer σ_p and of the property name σ_f. The intuition is that the pointer identifies the record, whereas the pointer and the property name identifies the property. In case the property was already present (RECUP-1) the structure of the record does not change and no demand is placed on the structure label. The new value, however, is raised to the write context of the property, which places the demand that the write context is below the label of the previous value. The existence label of the property is set to the greatest lower bound of the write context and the previous existence label. This allows the existence to be lowered when properties are written to in public context.

In case a new property is added (RECUP-2) the structure of the record changes, which places the demand that the write context of the record is below the structure label. As in the previous rule the value is raised to the write context of the property as is the existence label.

This prevents implicit flows, but stops the execution in the case the demands are not met as illustrated by the examples in the introduction.

Conditional branches. When a conditional branch (IF) is executed the label of the guard is used to elevate the pc for the body of the conditional. The pc is included in the security context of record update in the rules for variable and property assignment (VASSIGN and PASSIGN). The extent of the effect of the pc on the security context is illustrated by the box labeled pc in the figure to the right.



In case a conditional branch causes an elevated context (IF-H) the static component is applied to find and upgrade the

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, \dot{v}\rangle \\ \dot{p}_s, \sigma = \mathrm{find}(\!|h_2,\gamma_2,x|\!) \quad h_3 = h_2[\![(\dot{p}_s)[x^L] \xleftarrow{pc\sqcup\eta_2} \sigma]\!] \\ \dot{p} = \mathrm{find}(h_3,\gamma_2,x) \quad h_4 = h_3[(\dot{p})[x^L] \xmapsto{pc\sqcup\eta_2} \dot{v}]\end{array}}{\langle E_1, x := e\rangle \xrightarrow{pc} \langle\langle h_4,\gamma_2,\eta_2\rangle, \dot{v}\rangle}\ \text{VASSIGN}$$

$$\frac{\begin{array}{c}\langle E_1, e_1\rangle \xrightarrow{pc} \langle E_2, \dot{p}\rangle \\ \langle E_2, e_2\rangle \xrightarrow{pc} \langle E_3, \dot{f}\rangle \\ \langle E_3, e_3\rangle \xrightarrow{pc} \langle\langle h_4,\gamma_4,\eta_4\rangle, \dot{v}\rangle \\ h_5 = h_4[(\dot{p})[\dot{f}] \xmapsto{pc\sqcup\eta_4} \dot{v}]\end{array}}{\langle E_1, e_1\,[e_2]\ := e_3\rangle \xrightarrow{pc} \langle\langle h_5,\gamma_4,\eta_4\rangle, \dot{v}\rangle}\ \text{PASSIGN}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, b^\sigma\rangle \quad \sigma \sqsubseteq pc \sqcup \eta_2 \\ \langle\langle h_2,\gamma_2,\eta_2\rangle, s_b\rangle \xrightarrow{pc\sqcup\sigma} \mathcal{C}\end{array}}{\langle E_1, \texttt{if}\ e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \xrightarrow{pc} \mathcal{C}}\ \text{IF-L}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, b^\sigma\rangle \quad \sigma \not\sqsubseteq pc \sqcup \eta_2 \quad \langle E_3 \sqcup E_4, s_b\rangle \xrightarrow{pc\sqcup\sigma} \mathcal{C} \\ \langle\langle h_2,\gamma_2,\eta_2\rangle, s_{\texttt{true}}\rangle \xRightarrow{pc\sqcup\sigma} E_3 \quad \langle\langle h_2,\gamma_2,\eta_2\rangle, s_{\texttt{false}}\rangle \xRightarrow{pc\sqcup\sigma} E_4\end{array}}{\langle E_1, \texttt{if}\ e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \xrightarrow{pc} \mathcal{C}}\ \text{IF-H}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle E_2, \langle \overline{x}, s, \gamma\rangle^{\sigma_1}\rangle \quad \langle E_2, \overline{e}\rangle \xrightarrow{pc} \langle\langle h_3,\gamma_3,\eta_3\rangle, \overline{v}\rangle \\ h_4 = h_3[p_1 \mapsto \langle\{\overline{x} \overset{L}{\mapsto} \overline{v}^{\sigma_2}\}, \sigma_2\rangle] \quad p_1 \text{ fresh in } h_3 \\ h_5 = h_4[p_2 \mapsto \langle\{\mathrm{vars}(s) \overset{L}{\mapsto} \texttt{null}^{\sigma_2}\}, \sigma_2\rangle] \quad p_2 \text{ fresh in } h_4 \\ \gamma_4 = \langle \gamma_3, p_1^L\rangle \quad \gamma_5 = \langle \gamma_4, p_2^L\rangle \quad \sigma_2 = pc \sqcup \eta_3 \sqcup \sigma_1 \\ \langle\langle h_5,\gamma_5, pc\sqcup\eta_3\rangle, s\rangle \xrightarrow{pc\sqcup\sigma_1} \langle\langle h_6,\gamma_6,\eta_6\rangle, \dot{v}\rangle\end{array}}{\langle E_1, e\,(\overline{e})\,\rangle \xrightarrow{pc} \langle\langle h_6,\gamma_3,\eta_3\rangle, \dot{v}\rangle}\ \text{CALL}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2, \langle\gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, str^{\sigma_2}\rangle \\ s = \mathrm{parse}(str) \quad \overline{x} = \mathrm{vars}(s) \quad h_3 = h_2[p \mapsto o] \\ o = \mathrm{declare}(\overline{x}, pc \sqcup \eta_2 \sqcup \sigma_1 \sqcup \sigma_2, h_2[p]) \\ \langle\langle h_3, \langle\gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, s\rangle \xrightarrow{pc\sqcup\sigma_2} \mathcal{C}\end{array}}{\langle E_1, \texttt{eval}\ e\rangle \xrightarrow{pc} \mathcal{C}}\ \text{EVAL}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, \dot{p}\rangle \\ \langle\langle h_2, \langle\gamma_2, \dot{p}\rangle, \eta_2\rangle, s\rangle \xrightarrow{pc} \langle h_3,\gamma_3,\eta_3\rangle\end{array}}{\langle E_1, \texttt{with}\ e\ s\rangle \xrightarrow{pc} \langle h_3,\gamma_2,\eta_2\rangle}\ \text{WITH} \qquad \frac{pc \sqsubseteq \eta \quad \langle\langle h,\gamma,\eta\rangle, e\rangle \xrightarrow{pc} \langle E, \dot{v}\rangle}{\langle\langle h,\gamma,\eta\rangle, \texttt{return}\ e\rangle \xrightarrow{pc} \langle E, \dot{v}^\eta\rangle}\ \text{RETURN}$$

$$\frac{\begin{array}{c}\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle h_2,\gamma_2,\eta_2\rangle \\ \langle\langle h_2,\gamma_2,\eta_2\rangle, s_2\rangle \xrightarrow{pc} \mathcal{C} \quad \eta_2 \sqsubseteq \eta_1 \sqcup pc\end{array}}{\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1 \texttt{;}\ s_2\rangle \xrightarrow{pc} \mathcal{C}}\ \text{SEQ-CONT}$$

$$\frac{\begin{array}{c}\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle h_2,\gamma_2,\eta_2\rangle \quad \langle\langle h_2,\gamma_2,\eta_2\rangle, s_2\rangle \xRightarrow{pc} E_3 \\ \langle E_3, s_2\rangle \xrightarrow{pc} \mathcal{C} \quad \eta_2 \not\sqsubseteq \eta_1 \sqcup pc\end{array}}{\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1 \texttt{;}\ s_2\rangle \xrightarrow{pc} \mathcal{C}}\ \text{SEQ-CONT-H}$$

$$\frac{\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, \dot{v}\rangle \quad \eta_2 \sqsubseteq \eta_1 \sqcup pc}{\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1 \texttt{;}\ s_2\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, \dot{v}\rangle}\ \text{SEQ-HALT}$$

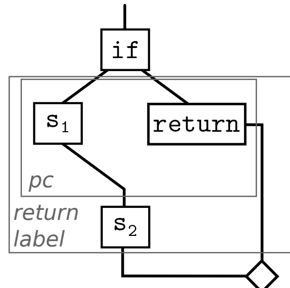
$$\frac{\begin{array}{c}\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1\rangle \xrightarrow{pc} \langle\langle h_2,\gamma_2,\eta_2\rangle, \dot{v}\rangle \quad \eta_2 \not\sqsubseteq \eta_1 \sqcup pc \\ \langle\langle h_2,\gamma_2,\eta_2\rangle, s_2\rangle \xRightarrow{pc} E_3\end{array}}{\langle\langle h_1,\gamma_1,\eta_1\rangle, s_1 \texttt{;}\ s_2\rangle \xrightarrow{pc} \langle E_3, \dot{v}\rangle}\ \text{SEQ-HALT-H}$$

Fig. 4: Selected hybrid monitor rules

labels of potential write targets in both branches before actually executing the body in the join of the static results. Otherwise, (IF-L) execution proceeds without using the static component.

The join of the environments $E_1 \sqcup E_2$ is defined structurally over the two environments point-wise joining the security labels. This is possible, since the static component only changes labels and never values. Thus, E_3 and E_4 in IF-H are guaranteed to have the same values and structure. For space reasons the full definition can be found in the full version of this paper [1]. Also note that the static component does not change the scope chain, only the records that the scope chain refers to.

Return. While the pc follows the syntactic structure of the program, returning from a secret context has the effect of extending the secret context to the end of the function as illustrated in the figure to the right.



In line with the base-line dynamic monitor [26], [27], [29] we employ a return label η to handle the non-syntactic control flow arising from return. This approach scales to other sources of non-syntactic control flow, see Section IV-D.

The return label is part of the execution environment and follows the call stack of the program: each function call has its own return label as defined in the rule for function call (CALL). In the same way as the pc, the return label is included in the write context by the rules for variable and property assignment (VASSIGN and PASSIGN).

In the semantics for `return` (RETURN) it is demanded that the return label is not below the pc. This prevents implicit flows via side effects in the part of the function under indirect control of the `return`, but stops the execution in case the demand is not met, see Example 6.

To increase permissiveness the static component will increase the return label if any return statements are found while analyzing a secret context. In turn, this leads to an elevated context for all following statements (SEQ-CONT-H), and, similar to conditional branches, the static component is applied. In the case the `return` is executed, control will be passed to the end of the function body and the statements syntactically after the return will not be executed. However, it is important that the static analysis is applied regardless (SEQ-HALT-H). Otherwise, the application of the static component would depend on secrets and, hence, also any upgraded security labels. In the case the return context is not raised, the static component is not applied (SEQ-CONT and SEQ-HALT).

To illustrate the interplay between the return label, statement sequence, and assignments consider Example 6 under the

$$\frac{\mathrm{has}(h[p], x) = \mathtt{true}^{\sigma_2}}{\mathrm{find}(h, \langle \gamma, p^{\sigma_1} \rangle, x) = p^{\sigma_1 \sqcup \sigma_2}}\ \textsc{Find-1} \qquad \frac{\mathrm{has}(h[p], x) = \mathtt{false}^{\sigma_2}}{\mathrm{find}(h, \langle \mathtt{null}, p^{\sigma_1} \rangle, x) = p^{\sigma_1 \sqcup \sigma_2}}\ \textsc{Find-2} \qquad \frac{\mathrm{has}(h[p_t], x) = \mathtt{false}^{\sigma_2} \quad \mathrm{find}(h, \gamma, x) = p_c^{\sigma_3}}{\mathrm{find}(h, \langle \gamma, p_t^{\sigma_1} \rangle, x) = p_c^{\sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}}\ \textsc{Find-3}$$

$$\frac{\mathrm{has}(h[p], x) = \mathtt{true}^{L}}{\mathrm{find}\llparenthesis h, \langle \gamma, p^{L} \rangle, x \rrparenthesis = p, L}\ \textsc{SFind-1} \qquad \frac{\mathrm{has}(h[p], x) = \mathtt{false}^{\sigma}}{\mathrm{find}\llparenthesis h, \langle \mathtt{null}, p^{L} \rangle, x \rrparenthesis = p, \sigma}\ \textsc{SFind-2} \qquad \frac{\mathrm{has}(h[p_t], x) = b^{\sigma_2} \quad \mathrm{find}\llparenthesis h, \gamma, x \rrparenthesis = p_c, \sigma_3}{\mathrm{find}\llparenthesis h, \langle \gamma, p_t^{\sigma_1} \rangle, x \rrparenthesis = p_c, \sigma_1 \sqcup \sigma_2 \sqcup \sigma_3}\ \textsc{SFind-3}$$

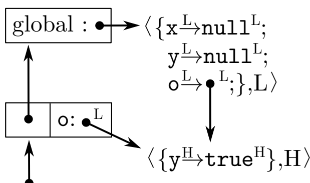
Fig. 5: Find and static find

assumption that `h` is `true`. The secret conditional causes an elevated security context and the static component is applied to the body (IF-H) to compute the upgraded environment in which the selected branch will be run. Due to the return under secret control (S-RETURN) the return label of this environment is H. This guarantees the success of the execution of the conditional (RETURN), but also means that in the sequence `if (h) { return true; }; x = 1; return false;` the return context is elevated after the conditional. This causes the static component to be applied to the rest of the function (SEQ-HALT-H), and even if `x = 1;` is not executed the static component will upgrade the outer variable `x` (S-VASSIGN).

Variable assignment. The scope chain is a sequence of dynamic records, called binding records, whose properties represent the defined variables. There are two constructions that introduce new scopes in the scope chain. First, function call (CALL) introduces two new scopes; one for the parameters, and one for the local variables of the function. Second, the `with` statement (WITH) injects a user defined record as a binding record in the scope chain. In both cases, the new scope is linked with the rest of the scope chain.

When executing an assignment (VASSIGN), the scope chain is searched to find the target of the write. Variable lookup is done by $\mathrm{find}(\cdot)$ (Figure 5) and is performed by starting at the top of the scope chain looking for the first binding record that binds the variable. A labeled pointer to the binding record is returned. The label of the pointer can be interpreted as whether existence of the variable in the corresponding binding record is secret, and is accumulated while traversing the chain (the absence of the variable in previous records can be inferred from knowing the presence in later records).

The result of $\mathrm{has}(\cdot)$ is labeled with the existence security label of the property in case the property exists and the structure security label otherwise, i.e., $\mathrm{has}(\langle \phi, \varsigma \rangle, x)$ evaluates to $\mathtt{true}^{\sigma}$, when $x \overset{\sigma}{\mapsto} v \in \phi$ and to $\mathtt{false}^{\varsigma}$ otherwise. Hence, the presence of binding records with secret structure in the scope chain will potentially cause variable assignment to stop. This will happen when the target is a public variable shadowed by a structurally secret record as illustrated in the figure to the left, depicting the scope chain before the execution of `x=true` in the body of the `with` in the following example.




```
var x; var y; var o = {  }; if (h) {o["y"] = true;};
with (o) {x = true; y = true;}
```

For this reason variable assignment uses static find, $\mathrm{find}\llparenthesis\cdot\rrparenthesis$ (Figure 5), to find the first public write target, x in p_s, and its accumulated scope context, σ.

Thereafter, static update, $h[\![(\dot{p})[\dot{x}] \overset{\sigma}{\leftarrow} ctx]\!]$ (Section IV-C), is used to upgrade the potential write target to the scope context. This prevents the actual update from stopping due to the scope context. Thus, static find and update play the same role for variable update as the static component does for standard execution.

Returning to the figure above, writing to `y` will upgrade the label of the variable `y` even though the update is captured by the record injected by the `with`. Similarly, the label of `x` is upgraded before the write, which prevents the monitor from stopping. The upgrade of `y` illustrates that the label update is independent of secrets, while the upgrade of `x` illustrates the increased permissiveness in that the execution is not stopped.

Function call. Function call (CALL) computes a function closure and calls the body in a new environment binding the given parameters to their formal names. Following JavaScript, variable declarations in the body of a function call are hoisted into the new environment separate from the parameter environment.

For function calls the label of the closure is part of the *pc* of the execution. The need for this is illustrated by the following example that copies the value of `h` to `x` by selecting different functions.

```
var x; var f;
if (h) { f = function() { x = true; }; }
else   { f = function() { x = false; }; }
f();
```

Thus, calling a secret function may cause an elevated write context. Unfortunately, applying the static component in such cases is not sound, since it makes the result of the static component directly depend on secrets. It is worth noting that it is for this reason all newly declared parameters and variables are labeled in accordance with the write context. Otherwise, writing to local variables in the body of a secret function would cause a security exception.

It is, however, sound to trace function calls inside elevated contexts. Consider Example 7, where the `set` function is called from within a elevated context. When reaching the secret conditional the static component is applied (IF-H) before executing the chosen branch. In this case, thanks to the ability to make use of runtime values, the static component (S-CALL-L) is able to identify the called function, analyze the body and update the write target, `data` (S-VASSIGN).

Eval. `eval` (EVAL) computes a string, parses it and executes the result in the environment of the caller. Similar to function calls, variable declarations in the program represented by the string given to `eval` are hoisted. For `eval` this is done using $\mathrm{declare}(\cdot)$ (Figure 6) that declares any undeclared variables given that write context is below the structure security level of the topmost binding record. In case no variables are declared no demand is placed on the structure label.

The label of the string providing the program to be evaluated is part of the *pc* of the execution. The above example is readily adapted to the setting of `eval` as follows.

$$\text{declare}(\overline{x}, ctx, \langle \phi, \varsigma \rangle) = \text{ let } X = \overline{x} \setminus dom(\phi) \text{ in}$$
$$\begin{cases} \langle \phi \cup \{X \overset{\varsigma}{\mapsto} \texttt{null}^{\varsigma}\}, \varsigma \rangle & \text{when } ctx \sqsubseteq \varsigma \\ \langle \phi, \varsigma \rangle & \text{when } X = \emptyset \end{cases}$$

Fig. 6: declare

```
var x; var s;
if (h) { s = "x = true;"; } else { s = "x = false;"; }
eval(s);
```

Evaluating a secret string is subject to the same limitations as calling a secret function. In the same way it is sound to analyze `eval` inside elevated contexts. Consider Example 8, where `eval` is called from within a elevated context. When reaching the secret conditional the static component is applied (IF-H) before executing the chosen branch. In this case, the static component is able to compute the string passed to eval and analyze the parsed result (S-EVAL-L), which causes the structure of the local variable environment to become secret. Now, regardless of whether the branch is taken or not the assignment `l = false` causes the outer variable l to be upgraded (V-ASSIGN) as discussed above.

C. Static component

The static component is applied before all elevated write contexts in order to find potential write targets and update their security labels to prevent the execution from being stopped for security reasons. The static component consists of static versions of the statement and expression semantic as well as static versions of other semantic parts that cause or influence side effects (e.g., find and record update).

Since the hybrid analysis is based on a sound dynamic analysis is suffices that the static component does not violate the invariants of the dynamic analysis: 1) it must not make the labeling less secret, and 2) the labeling must remain independent of secrets. In particular, the static component does not need to be complete in the sense that all potential write targets are found. Missing a write target may affect the permissiveness of the hybrid monitor but does not jeopardize the soundness, since it is guaranteed by the base-line dynamic monitor. This is important because it allows the static component to ignore potential write targets (e.g., when the target cannot be precisely established) instead of being overly conservative. As an example, consider the presence of operations that are hard to compute statically, e.g., operations on strings with the result potentially fed into `eval`.

Further, the static component is *value sensitive* in the sense that it makes use of runtime values. This is of decisive importance for being able to handle dynamic constructs like records, first class functions and the scope chain. For instance, Example 7 would not be possible to handle without value sensitivity, since the static component must have knowledge of the called closure.

The static statement component is of the form $\langle E_1, s\rangle \overset{pc}{\Longrightarrow} E_2$, read as: analysis of statement s before execution under the program counter pc, in environment E_1 results in a relabeling of E_1 denoted as E_2. The static statement component is similar to the dynamic monitor with three key differences: 1) side effects are limited to labels only and the new labels are at least as secret as the old labels, 2) all execution paths are analyzed and secret values are not used 3) the static component must be able to handle unknown values. The first difference corresponds to the demand that labels are not lowered by the analysis. In addition it makes it possible to execute the static analysis on the actual environment without changing the semantics of the program. The second difference corresponds to the demand that label changes are independent of secrets. Finally, the third difference comes from the need to statically compute values of expressions in order to resolve dynamic write targets. In many cases, this is possible; in general it is not.

Values. The values of the language are extended with a distinguished unlabeled unknown value, $\bullet$, that is added to the set of values v, the set of literals l and the set of pointers p. All operations are lifted accordingly. The lifting is simple: unknown values are treated as secrets and if the presence of an unknown value prevents the computation the result is $\bullet$. Consider, e.g., the lifting of $\text{find}(\cdot)$ to the extended values by the addition of a rule to handle unknown values is the scope chain as follows:

$$\frac{\text{find}(h, \gamma, x) = \dot{p}}{\text{find}(h, \langle \gamma, \bullet \rangle, x) = p^H} \text{ FIND-4}$$

For brevity, we omit explicitly lifting in the following. The full set of rules is found in the full version of this paper [1].

The unknown values are unlabeled reflecting that their labels are also unknown. However, matching p^σ with $\bullet$ is possible and will make $p = \bullet$ and $\sigma = H$.

Static record update. The core of the static component is performed by static record update. Concisely, static record update, $h[\![(\dot{p})[\dot{f}] \overset{pc}{\longleftarrow} ctx]\!]$, updates the label of the property f in the record pointed by p in heap h to ctx; if the property does not exist in the record the structure label of the record is updated instead. With the exception of the rule for `return` (S-RETURN) all label updates performed by the static component are done by static record update.

The productive rules for static record update are S-RECUP-L, S-RECUP-H and S-RECUP-S.

$$\frac{\langle \phi_1, \varsigma \rangle = h[p] \quad f \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1 \qquad pc \sqsubseteq \sigma_o \quad \phi_2 = \phi_1[f \overset{\sigma}{\mapsto} v^{\sigma_o \sqcup ctx}]}{h[\![(p^L)[f^L] \overset{pc}{\longleftarrow} ctx]\!] = h[p \mapsto \langle \phi_2, \varsigma \rangle]} \text{ S-RECUP-L}$$

$$\frac{\langle \phi_1, \varsigma \rangle = h[p] \quad pc \sqsubseteq \varsigma \quad \forall x \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1 \;.\; pc \sqsubseteq \sigma_o \qquad \phi_2 = \{x \overset{\sigma}{\mapsto} v^{\sigma_o \sqcup ctx} \mid x \overset{\sigma}{\mapsto} v^{\sigma_o} \in \phi_1\}}{h[\![(p^L)[f^H] \overset{pc}{\longleftarrow} ctx]\!] = h[p \mapsto \langle \phi_2, \varsigma \sqcup ctx \rangle]} \text{ S-RECUP-H}$$

$$\frac{\langle \phi, \varsigma \rangle = h[p] \quad f \notin dom(\phi) \quad pc \sqsubseteq \varsigma}{h[\![(p^L)[f^L] \overset{pc}{\longleftarrow} ctx]\!] = h[p \mapsto \langle \phi, \varsigma \sqcup ctx \rangle]} \text{ S-RECUP-S}$$

Rule S-RECUP-L updates an existing property to the given context under the condition that the pointer and the property names are known and public. If this was not the case the resulting label would be depending on more secret information. Unlike the dynamic monitor the static inspection does not stop if this is not the case; rather no updates are performed. Rule S-RECUP-H allows for updates over secret or unknown property names. Instead of not performing the update, the labels of all the properties of the entire record are updated as well as the structure label of the record. Finally, S-RECUP-S updates the structure of a record; since we are not performing side effects the property should not be added, but the structure should

$$\frac{\langle E_1, e_1\rangle \overset{pc}{\Longrightarrow} \langle E_2, \dot{p}\rangle \quad \langle E_2, e_2\rangle \overset{pc}{\Longrightarrow} \langle E_3, \dot{f}\rangle \qquad \langle E_3, e_3\rangle \overset{pc}{\Longrightarrow} \langle\langle h_4, \gamma_4, \eta_4\rangle, \dot{v}\rangle \quad h_5 = h_4[\![(\dot{p})[\dot{f}] \overset{L}{\longleftarrow} pc \sqcup \eta_4]\!]}{\langle E_1, e_1\,\texttt{[}e_2\texttt{]} \;\texttt{:=}\, e_3\rangle \overset{pc}{\Longrightarrow} \langle\langle h_5, \gamma_4, \eta_4\rangle, \dot{v}\rangle}\ \textsc{S-PAssign}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle \qquad \dot{p}, \sigma = \mathrm{find}(\!|h_2, \gamma_2, x|\!) \quad h_3 = h_2[\![(\dot{p})[x^L] \overset{L}{\longleftarrow} pc \sqcup \eta_2]\!]}{\langle E_1, x \;\texttt{:=}\, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_3, \gamma_2, \eta_2\rangle, \dot{v}\rangle}\ \textsc{S-VAssign}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle E_2, \langle \overline{x}, s, \gamma\rangle^L\rangle \quad \langle E_2, \overline{e}\rangle \overset{pc}{\Longrightarrow} \langle\langle h_3, \gamma_3, \eta_3\rangle, \overline{v}\rangle \\ h_4 = h_3[p_1 \mapsto \langle\{\overline{x} \overset{L}{\longmapsto} \overline{v}\}, L\rangle] \quad p_1 \text{ fresh in } h_3 \\ h_5 = h_4[p_2 \mapsto \langle\{\mathrm{vars}(s) \overset{L}{\longmapsto} \texttt{null}^L\}, L\rangle] \quad p_2 \text{ fresh in } h_4 \\ \gamma_4 = \langle \gamma_3, p_1^L\rangle \quad \gamma_5 = \langle \gamma_4, p_2^L\rangle \\ \langle\langle h_5, \gamma_5, \eta_3\rangle, s\rangle \overset{pc}{\Longrightarrow} \langle h_6, \gamma_6, \eta_6\rangle\end{array}}{\langle E_1, e\,\texttt{(}\overline{e}\texttt{)}\,\rangle \overset{pc}{\Longrightarrow} \langle\langle h_6, \gamma_3, \eta_3\rangle, \bullet\rangle}\ \textsc{S-Call-L}$$

$$\frac{\begin{array}{c}\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_2, \langle \gamma_2, p_1^L\rangle, \eta_2\rangle, str^L\rangle \\ s = \mathrm{parse}(str) \quad \overline{x} = \mathrm{vars}(s) \quad \langle \phi, \varsigma\rangle = h_2[p_1] \\ o = \mathrm{declare}(\!|\overline{x}, pc, \langle \phi, \varsigma\rangle|\!) \quad h_3 = h_2[p_1 \mapsto o] \\ h_4 = h_3[p_2 \mapsto \langle\{\overline{x} \setminus dom(\phi) \overset{\varsigma}{\longmapsto} \texttt{null}^\varsigma\}, \varsigma\rangle] \quad p_2 \text{ fresh in } h_3 \\ \gamma_3 = \langle\langle \gamma_2, p_1^L\rangle, p_2^L\rangle \quad \langle\langle h_4, \gamma_3, \eta_2\rangle, s\rangle \overset{pc}{\Longrightarrow} \langle h_5, \gamma_4, \eta_3\rangle\end{array}}{\langle E_1, \texttt{eval}\ e\rangle \overset{pc}{\Longrightarrow} \langle h_5, \langle \gamma_2, p_1\rangle, \eta_3\rangle}\ \textsc{S-Eval-L}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle E_2, v^H\rangle \quad \langle E_2, \overline{e}\rangle \overset{pc}{\Longrightarrow} \langle E_3, \overline{v}\rangle}{\langle E_1, e\,\texttt{(}\overline{e}\texttt{)}\,\rangle \overset{pc}{\Longrightarrow} \langle E_3, \bullet\rangle}\ \textsc{S-Call-H}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_2, \langle \gamma_2, p^{\sigma_1}\rangle, \eta_2\rangle, str^{\sigma_2}\rangle \quad \sigma_1 \sqcup \sigma_2 = H}{\langle E_1, \texttt{eval}\ e\rangle \overset{pc}{\Longrightarrow} E_2}\ \textsc{S-Eval-H}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{v}\rangle}{\langle E_1, \texttt{return}\ e\rangle \overset{pc}{\Longrightarrow} \langle h_2, \gamma_2, \eta_2 \sqcup pc\rangle}\ \textsc{S-Return}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle\langle h_2, \gamma_2, \eta_2\rangle, \dot{p}\rangle \qquad \langle\langle h_2, \langle \gamma_2, p\rangle, \eta_2\rangle, s\rangle \overset{pc}{\Longrightarrow} \langle h_3, \gamma_3, \eta_3\rangle}{\langle E_1, \texttt{with}\ e\ s\rangle \overset{pc}{\Longrightarrow} \langle h_3, \gamma_2, \eta_3\rangle}\ \textsc{S-With}$$

$$\frac{\langle E_1, e\rangle \overset{pc}{\Longrightarrow} \langle E_2, \dot{b}\rangle \qquad \langle E_2, s_{\texttt{true}}\rangle \overset{pc}{\Longrightarrow} E_3 \quad \langle E_2, s_{\texttt{false}}\rangle \overset{pc}{\Longrightarrow} E_4}{\langle E_1, \texttt{if}\ e\ s_{\texttt{true}}\ s_{\texttt{false}}\rangle \overset{pc}{\Longrightarrow} E_3 \sqcup E_4}\ \textsc{S-If}$$

Fig. 7: Selected static component rules

be raised to allow the property to be added by the dynamic monitor.

With this we are ready to investigate a selection of the most interesting static analysis rules (Figure 7). The remaining rules can be found in the full version of this paper [1].

Assignment. Static variable update and static property update are similar in that the potential write target is identified and upgraded using static record update.

For variable assignment (S-VASSIGN), the write target is identified using static find, $\mathrm{find}(\!|\cdot|\!)$. The reason static find is used rather than standard find, $\mathrm{find}(\cdot)$, is that the latter would not be meaningful. In case the scope context is public $\mathrm{find}(\!|\cdot|\!)$ and $\mathrm{find}(\cdot)$ result in the same write target, and in case the scope context is secret we cannot safely use the result of $\mathrm{find}(\cdot)$ to update. In particular, the label of the pointer returned by find would be secret which would prevent static record update from making any modifications. Rather, by using $\mathrm{find}(\!|\cdot|\!)$ we update the first potential write target that can safely be updated.

As for the actual upgrade, static record update is used to upgrade the write target to the write context. Note that the scope context is not taken into account in the upgrade — it is the responsibility of the hybrid assignment rule (VASSIGN) in case the assignment is actually run. See Section IV-B.

In the case of property update (S-PASSIGN) the write target is recursively computed and static record update is used to update the write target to the write context.

Conditional branches. The static inspection of statements follows the rules of the dynamic monitor with the difference that all potential paths are explored. For instance, the rule for conditional branches (S-IF) analyses both the *then* branch and the *else* branch regardless of the value of the guard.

With. The rule for `with` (S-WITH) injects the result of the expression whether it is known or not. This might inject unknown values into the scope chain, which is handled by $\mathrm{find}(\!|\cdot|\!)$.

Function call. Function call corresponds to the dynamic monitor if the closure to be called can be computed and is public. In such case, new temporary scope records are allocated for the call and the body of the function is analyzed. In the case the closure is unknown or secret no attempt at analyzing the body of is made. For recursive functions and iteration we employ standard techniques [41] for fixpoint computation.

The possibility to ignore cases, where we cannot establish which function was called, stems from that we rely on the static component only for permissiveness and not for soundness. Omitting the analysis of a function call will prevent the static component from identifying and upgrading the potential write targets of the call, which can cause the hybrid monitor to stop in case such a write is actually performed.

Eval. Static `eval` (S-EVAL-L) differs somewhat from the dynamic counterpart. If the string can be computed and is public, the parsed string can be analyzed. First, static hoisting is performed. Similar to records (S-RECUP-S) we cannot actually add variables to the context, since this may change the semantics of the program. Instead, the static declaration, $\mathrm{declare}(\!|\cdot|\!)$, upgrades the structure label of the topmost binding record in order to allow the execution of `eval` to hoist.

$$\mathrm{declare}(\!|\overline{x}, pc, \langle \phi, \varsigma\rangle|\!) = \begin{cases} \langle \phi, \varsigma \sqcup pc\rangle & \text{when } \overline{x} \setminus dom(\phi) \neq \emptyset \\ \langle \phi, \varsigma\rangle & \text{otherwise} \end{cases}$$

Second, an additional local binding record is created to hold the variables that would be hoisted into the environment of the caller during execution. The reason for this is to make sure that variables are properly captured during the static evaluation of the evaluated program. Consider the following program.

```
var x;
(function() {
  if (h) { eval("var x; x = 1"); }
})();
```

Unless the local environment was introduced, the static component would infer that the outer variable `x` was written, when actually captured by the variable defined by the evaluated program.

Return. The static rule for `return` (S-RETURN) increases the return label to the *pc*, which guarantees that returns statements will not cause a security errors. For an explanation on the interplay between the rules related to the return label see Section IV-B.

D. From the core to full JavaScript

JavaScript as defined by the ECMA-262 (v.5) [19] is beyond the scope of this paper. Instead we envision the current work to provide the theoretical basis for a scaling to full JavaScript mirroring the path taken by related efforts: from theory of dynamic information flow control for JavaScript [29] to practice [27]. We believe the effort to be roughly comparable with one exception: the standard API. Properly handling the standard API would require a hybrid model, which we leave as future work. The fact that the static component is only used to increase the precision of the hybrid analysis and not its soundness allows for important flexibility when scaling the analysis to the full language. With respect to the API this entails that it is possible to develop a working prototype for the entire language without modeling the API at the cost of precision. This would be no different than cases where it cannot be established which function is called. Other language constructions that benefit from this flexibility are, e.g., exceptions and implicit coercions.

Object creation and the prototype hierarchy. JavaScript uses prototype based inheritance. The prototype hierarchy is constructed on object creation by copying the contents of the constructor's `prototype` property to the internal prototype property of the newly created object. Since the prototype itself might have a prototype, the prototype hierarchy forms a chain of objects similar to the scope chain. When a property is accessed on an object a prototype chain lookup is performed: if the property is not present in the object itself, the lookup continues recursively through the prototype hierarchy until found or the hierarchy ends.

When combined with `with` the scope chain offers exactly the same challenges as the prototype hierarchy: the traversal of a chain of dynamic records searching for a certain property. In addition to modeling the prototype hierarchy full support for object creation requires modeling the `new` operation, which is analogous to function call.

Non-syntactic transfer of control. There are three constructions in JavaScript that allow for non-syntactic transfer of control: 1) exceptions, 2) `break` and `continue`, in particular together with labeled statement and, 3) the `return` statement.

The `return` statement transfers control to the end of a function, and `break` and `continue` statements interrupt the standard control flow of loops and `switch` statements.

Similarly, exceptions provide a way of non-syntactic transfer of control from the source of the exception to an exception handler in case one exists. Exceptions are different from the two other constructions in that they 1) allow for transfer across function calls, and 2) can be cause by primitive operations of the language as well as its API.

The handling of non-syntactic transfer of control is similar across the different constructions. In the current paper we introduce and explain the return label as the label of the maximum pc in which `return` is allowed. In case the return label is below the control context when reaching a `return` statement execution is stopped with an error. In the same way each labeled statement is associated with a security label that controls that is the maximum pc in which `break` and `continue` to the label are allowed, and similarly for exceptions an exception label is used. Like the return label, the statement security labels and the exception label form the write context together with the pc, see [27] for more information.

From a hybrid perspective the handling of the different labels is analogous; in case an instruction that causes non-syntactic transfer of control is found by the static component the corresponding label is raised. Following [27] it is reasonable to not raise the label for internal exceptions. This is a design choice that avoids the potentially drastic increase of secret control contexts at the cost of not allowing internal exceptions under high control. It made possible by the fact that the static component is only used to increase the permissiveness of the analysis.

It might be worth pointing out that Hedin and Sabelfeld [29] exemplify non-syntactic transfer of control using exceptions, while assuming that functions have a *unique exit point*. This assumption allows them to simplify the rule for `return`. In their subsequent work [27] a return label similar to the one presented in this paper is used. To keep the language small we have opted to use the return label to represent the handling of non-syntactic transfer of control.

While this approach scales the other constructions of JavaScript that cause non-syntactic control flow its practical permissiveness on wild JavaScript must be evaluated. Both Just et al. [33] and Bichhawat et al. [10] argue for the need for control-flow analysis to handle non-syntactic (unstructured in their terminology) control flow. In a sense, in combination with the static component, the label approach can be seen as a limited local control-flow analysis empowered by runtime values and constitutes a natural first step given the base-line dynamic monitor. However, if needed, thanks to the modularity of the hybrid monitor, it is possible to further strengthen the handling of non-syntactic control flow to more advanced and precise control-flow analyses at the expense of performance.

Accessor properties and property attributes. JavaScript allows the programmer to associate functions to properties that are called when the property is read or when a value is assigned to the property. From an information flow perspective accessor properties offer challenges that are similar to those offered by function valued properties, with the difference that the associated functions are called on reading and writing [27]. Similarly, JavaScript associates a number of property attributes to properties. These attributes control different aspects of the property, e.g, if the property can be deleted, or if it is enumerable. Modeling property attributes is direct; each property is extended with security labeled attributes in addition to the value or the accessor functions. Tracking information flow into attributes does not differ from tracking information flow for standard values from both a purely dynamic and a hybrid perspective.

Implicit coercions. Implicit coercions may give rise to complex information flow [27]. From a purely dynamic perspective implicit coercions are relatively easy to handle, whereas from a hybrid perspective tracing implicit coercions in the static component may introduce a lot of potential flows that will never occur during execution. Like for internal exceptions, it may be reasonable not to track flows due to implicit coercions in the static component. This will not jeopardize soundness, which is guaranteed by the baseline dynamic monitor. It might, however, cause the hybrid analysis to stop with a security error in case the implicit coercion was actually used and resulted in the upgrade of a security label under secret control.

E. Practical considerations

When deploying runtime analyses in practice the execution overhead brought by the analysis is important. While the current paper is aimed at developing a hybrid analysis that *enables* runtime information flow analysis of real programs, making the analysis practically useful is an important part of the long term goal.

Once the hybrid approach has been deemed viable from a permissiveness perspective it remains to make it practically useful. This entails extending an existing implementation, ideally one of the state-of-the-art implementations, that can serve as a baseline for comparison, and optimize the hybrid monitor by, for instance, utilizing static pre-computation of potential write locations. Without a baseline implementation, the comparison is one where an experimental unoptimized implementation is contrasted to a highly optimized commercial implementation. Such comparison risks significantly overestimating the actual overhead of hybrid information flow enforcement.

V. Soundness

This section establishes that our enforcement mechanism guarantees the baseline security condition of *termination-insensitive noninterference* [22], [46], [55]. The intuition behind noninterference is simple: all attacker observables must be independent of any secret information given to the program. This is typically phrased in terms of pairs of executions of the program. If, for any two executions where the public information is kept the same but where the secret information is allowed to be different, the attacker observables remain the same then they are independent of the secret information.

In the web setting, examples of attacker observables could be sending or retrieving information over the network. In general, attacker observables provide a partial view of the execution environment and, hence, security notions for attacker observables are subsumed by security notions for the execution environment.

Noninterference can be phrased as the preservation of a family of low-equivalence relations $\sim_\beta$ under execution, where low-equivalence captures the notion of keeping the public information the same while allowing the secret to vary. The family is indexed by a bijection on the low-reachable domain of the heaps, which guarantees that the public heap structure is isomorphic. Intuitively, two environments E_1 and E_2 are low-equivalent if all the public information contained within them are equal. With this we can formulate the toplevel security condition for programs, s.

Theorem 1. *Soundness of the hybrid monitor*

$$E_1 {\sim_{\beta_1}} E_1' \land \langle E_1, s\rangle \xrightarrow{pc} E_2 \land \langle E_1', s\rangle \xrightarrow{pc} E_2' \Rightarrow \exists \beta_2 \;.\; \beta_1 \subseteq \beta_2 \land E_2 {\sim_{\beta_2}} E_2'$$

Proof: The proof proceeds by mutual induction on execution derivations and relies on confinement of statements and expressions as well as noninterference of the static component. We refer the reader to the full version of this paper [1] for details. ■

VI. Permissiveness

The fundamental goal of the hybrid monitor presented in this paper is to increase the permissiveness of purely dynamic monitors.

There are two potential sources of inaccuracies in the hybrid monitor: 1) underapproximation, i.e., when the static component fails to identify a write target under secret control, and 2) overapproximation, i.e., when the static analysis wrongly identifies a write target. As was established in Section IV neither jeopardize the soundness of the hybrid monitor, since it is guaranteed by the sound base line dynamic monitor. However, both may cause the hybrid monitor to halt secure programs with a security error.

To highlight advantages of the hybrid approach, we discuss how it handles common programming patters. Additionally, it has been previously shown that purely dynamic enforcement is incomparable to purely static enforcement [44]. For this reason it is interesting to compare our hybrid enforcement with both purely dynamic enforcement as well as purely static enforcement. Finally, we compare our approach to upgrade instructions.

Example	1	2	3	4	5	6	7	8
Dynamic	✗	✗	✗	✗	✗	✗	✗	✗
JSFlow	✓	✗	✗	✗	✗	✗	✗	✗
Type System	✓	✗	✓	✓	✗	✗	✗	✗
V-hybrid	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 8: Relative permissiveness

A. Comparison on common patterns

We illustrate the permissiveness of the hybrid analysis by comparing the different approaches in the light of the examples presented in the introduction. The examples originate from our experimentation with running real code and, as such, represent programs that we deem plausible to be found in the wild.

Let $\langle E, s\rangle \rightsquigarrow \mathcal{C}$ denote the dynamic monitor obtained by removing all uses of the static component in the hybrid monitor $\langle E, s\rangle \rightarrow \mathcal{C}$. Figure 8 contains an overview over how the different approaches compare on the examples. The rows represent the different approaches where *V-hybrid* (short for value-sensitive hybrid) is the hybrid monitor presented in this paper, *Dynamic* is the pure dynamic monitor presented above, *JSFlow* is the JavaScript dynamic monitor [27], and *Type System* is the flow-sensitive type system. In the figure, ✓ denotes that the example is accepted, and ✗ denotes that the example is rejected.

All the eight examples are accepted by our hybrid approach and rejected by the purely dynamic monitor. It is interesting to note that the simplistic hybrid variable approach of JSFlow allows for Example 1 but none of the other examples.

With respect to the flow-sensitive type system presented below only Example 1 can be handled in a flow-sensitive way. In addition Example 3 and Example 4 can be typed in a *flow-insensitive* way by making the initial record type sufficiently secret. Flow-sensitive typing of those examples is not possible, though, due to limitations of flow-sensitivity in the presence of aliases. For this reason, Example 2, and Example 5 cannot be handled by the type system; the initial record cannot be given any other type than the empty record type. Additionally, Example 6 and Example 7 both contain functions.

B. Versus pure dynamic monitors

First, we define relative permissiveness for monitors. The set of productive environments for a monitor, M, and program, s, $P_M(s)$ is defined as the environments for which the monitor does not stop (with a security error or otherwise), i.e., $P_M(s) = \{E \mid \exists \mathcal{C} \;.\; (\langle E, s\rangle, \mathcal{C}) \in M\}$. Thus $P_{\rightsquigarrow}$, and $P_{\rightarrow}$ are families of productive sets (productive families) indexed by programs for the pure dynamic monitor, and the hybrid monitor, respectively.

We say that a monitor M_1 is more permissive than a monitor M_2 if the productive family of M_2 is a subfamily of the productive family of M_1.

Theorem 2. *The pure dynamic monitor, $\rightsquigarrow$, is not more permissive than the hybrid monitor, $\rightarrow$, $P_{\rightsquigarrow} \not\subseteq P_{\rightarrow}$.*

Proof: Any of the examples in Section I are counterexamples to the converse statement. ■

While we have obtained the desired result, in line with our goal with the hybrid monitor, to increase the permissiveness for practical examples, due to the fundamental tension [44] between static and dynamic enforcement, the hybrid monitor does not subsume permissiveness of the pure dynamic monitor.

Theorem 3. *The hybrid monitor, $\rightarrow$, is not more permissive than the pure dynamic monitor, $\rightsquigarrow$, $P_{\rightarrow} \not\subseteq P_{\rightsquigarrow}$.*

Proof: The following program provides a counter example to the converse statement.

Example 9

```
var o; var p; var q;
p = { f : true }; q = { }; o = p;
if (h) { o = q; };
o["f"] = false;
```

In an environment where `h = false` the pure dynamic monitor will successfully execute the above program, whereas the hybrid analysis will not, since `o` will contain a secret pointer after the static execution of the conditional branch, which causes the last assignment to fail with a security error. ■

On the possibility of subsuming (sound) purely dynamic analyses. With the above definition of relative permissiveness a hybrid monitor cannot subsume a purely dynamic monitor. Consider for instance the following program containing *dead code*:

```
l = true; if (h != h) { l = h; }; out(l);
```

A purely dynamic analysis does not have to stop the above program, while a hybrid analysis will in case it analyzes the body of the conditional. Even if the hybrid analysis tries to detect dead code, in general, it is not possible, which thwarts subsumption.

Even in the absence of dead code, subsumption is not possible given that the security notion is termination-insensitive noninterference. Consider the following *insecure* program:

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

Despite leaking the entire secret to the attacker, a purely dynamic analysis will allow the program to execute in environments where `h` is `false`, whereas a hybrid analysis will detect the leak and (rightfully) stop in all environments. Hence, for insecure programs the dynamic monitor is potentially more permissive than the hybrid with respect to termination-insensitive noninterference, since it may allow the insecure programs to run in more environments.

C. Versus static analysis

We follow the approach of [44] and compare our enforcement to a typical static flow-sensitive analysis. The power of the language in our paper demands both precise *must-alias* and value information for static enforcement as does handling `with` and first class functions in the presence of flow-sensitivity. Such information is not present in a standard flow-sensitive type system. Thus, for this comparison the type system restricts the use of the language: 1) records are flow-insensitive, and projection and update are only allowed on statically decidable property names 2) functions are not supported, since they, due to dynamic scopes, need to be typed in the environment of the caller, which requires the type system to know which function is called 3) `eval` is not supported. These restrictions severely limits the use of the language; little of the original dynamism remains. In itself, the need to restrict the language is a strong argument for the hybrid approach.

Type language. To create a static type system for the language a type language is needed. Let σ range over security labels, let ω denote record types: maps from properties to primitive types In addition, let ν be variable record types defined in the same way as record types but with the difference that we know that variable records will be unaliased. Hence, variable record types are flow sensitive, whereas record types in general are not. Finally, Γ denotes the type of the scope chain: a sequence of record and variable record types that representing the records of the chain.

$$\tau ::= \sigma \,\big|\, \omega \quad \omega ::= \{\overline{x} : \overline{\tau}\} \quad \nu ::= \{\overline{x} : \overline{\tau}\}$$
$$\Gamma ::= \omega \cdot \Gamma \,\big|\, \nu \cdot \Gamma \,\big|\, \bot$$

In order to relate the type system to the hybrid monitor we need to tie the types to the values of the language. As is standard this is done via a well-formedness relation $\delta, \xi \vdash E : \Gamma$ defined structurally demanding that the runtime labels correspond to the static types. Note that the typing relation for pointers is split into a typing of pointers to general records, δ, and the typing of pointers to variable records, ξ. For space reasons the definition of the relation, the type rules and the proofs can be found in the full version of this paper [1].

Type system. The flow-sensitive type system for the restricted language has judgments of the form $pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2$ read the statement s is type correct in program counter pc, and scope chain type Γ_1 resulting in and environment with scope chain type Γ_2.

We prove two theorems that relate the hybrid monitor to the static flow-sensitive type system: *permissiveness* and *accuracy*.

The permissiveness theorem (Theorem 4) states that the hybrid monitor will not stop on well-typed programs when run in well-formed environments. For this, we must extend the monitor semantics to return a distinguished security error denoted $\triangledown$ when execution is stopped due to a security violation.

Theorem 4. *Permissiveness*

$$pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2 \wedge \delta, \xi \vdash E_1 : \Gamma_1 \wedge \langle E_1, s\rangle \xrightarrow{pc} E_2 \Rightarrow E_2 \neq \triangledown$$

Proof: By mutual induction on the execution derivation. ■

The accuracy theorem (Theorem 5) establishes that the type of the result of the execution monitored by our hybrid mechanism is not more secret than the type system. In other words, the security labeling produced is at least as accurate as the static type system.

Theorem 5. *Accuracy*

$$pc, \Gamma_1 \vdash s \Rrightarrow \Gamma_2 \wedge \delta_1, \xi_1 \vdash E_1 : \Gamma_1 \wedge \langle E_1, s\rangle \xrightarrow{pc} E_2 \Rightarrow \exists \delta_2, \xi_2 \,.\, \delta_2, \xi_2 \vdash E_2 : \Gamma_2$$

Proof: By mutual induction on the execution derivation. ■

Together with Figure 8 those results show that the hybrid monitor is strictly more permissive than the static type system.

D. Versus upgrade instructions

Birgisson, Hedin, and Sabelfeld [12] investigate how to use testing to automatically inject upgrade instructions into a program in order to alleviate the restrictions imposed by the NSU. They show how the approach can give better accuracy than static typing, especially when richer security lattices than the two level lattice is used. The idea is based on the fact that public labels are allowed to depend on public information. Consider, for instance,

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

Our hybrid monitor enjoys the same increase compared to static typing: in case `l` is `true` the label of `x` is `H`, but if `l` is `false` the label of `x` remains `L`.

An important realization made in [12] is that the semantics of upgrades needs to be relatively sophisticated. For instance, the notion of delayed upgrades is introduced in order to not upgrade values to early. Consider the following example (taken from [12])

```
o = {}; o["f"] = 1; x = o["f"]; if (h) { o["f"] = 42; }
```

The issue is that in general it is not possible to insert an upgrade instruction just before the conditional, since there might be no way at that point to syntactically refer to the same property (in the above program it is, but in general it might not be). Thus, the upgrade must be inserted at a previous write access (it must exist, see [12]). However, upgrading at `o[0] = 1` is not satisfactory, since it would make `x` secret. Thus, the solution of [12] is to introduce an upgrade instruction that delays the upgrade to a certain statement. For the constructions of the language of this paper even more advanced upgrade instruction must be added —for instance, there is no way to syntactically refer to variables in a closure, needed for Example 7.

The hybrid approach avoids this breed of problems altogether. Since the monitor has full access to the execution state it can perform any updates just before entering the context and the `f` property of `o` is upgraded before entering the conditional.

E. Versus permissive upgrades

Another way of improving the performance of dynamic information-flow monitors it *permissive upgrades* [5]. The approach allows the labels to change under secret control, but to disallow any future branching on the resulting value. This allows secret conditionals to be run, but severely limits the use of the resulting value — most operations involve some kind of branching on the value — and any derived values.

Our approach does not suffer from these restrictions. We are able to upgrade labels of potential write targets without putting restrictions on the resulting values.

F. Extension to declassification

While the hybrid approach of this paper increases the precision of dynamic information-flow enforcement there is a category of programs that contain information flow, but that we intuitively would like to classify as secure. The most common example of such programs is a password checker: if the password matches, the user is granted access and if not an error message is displayed.

```
if (password == guess) { result = true; }
else { result = false; }
```

While clearly leaking information about the password to the public login display, under certain circumstances we can classify the program as secure. One possible way to define these circumstances, known as semantic security, is to state that the program is secure if a polynomial time attacker has negligible probability of success (with respect to the length of the password) of acquiring the password. The password checker program can be shown [2] to be semantically secure for (large) passwords drawn uniformly at random. Such analysis is beyond the scope of traditional information-flow enforcement and beyond the approach of this paper.

Nevertheless, it is still illustrative to compare our hybrid monitor to a pure dynamic monitor on the above program. While a pure dynamic monitor would stop execution with a security error when trying to update the public variable `result` our monitor would correctly identify the flow and upgrade `result` to secret.

VII. RELATED WORK

A large, extensively surveyed [11], [28], [36], [46], body of work studies information-flow control. We focus on discussing sound hybrid information flow control and information flow control for JavaScript.

As mentioned earlier, formalization of hybrid mechanisms [35], [37], [44], [49] have been demonstrated to enforce noninterference, however for simple languages without dynamic structures as the heap.

Inlined information-flow security monitors can be viewed as hybrid monitors: these monitors are results of *static* program transformation that inlines *dynamic* security checks inside of the code of a given program. The inlined code manipulates shadow variables to keep track of the security labels of the program's variables. Inlining information-flow security monitors have been explored for simple languages [9], [14], [39] without the heap. This approach has been pushed in the direction of JavaScript [48] targeting a large language subset including the scope chain, the heap and prototypical inheritance as well as closures. In addition, the result is proved to satisfy termination insensitive noninterference given the proposed semantics. However, there are fundamental limits in the scalability of the shadow-variable approach. The execution of a vast majority of the JavaScript operations (with the prime example being the + operation) is dependent on the types of their parameters. This might lead to coercions of the parameters that, in turn, may invoke such operations as `toString` and `valueOf`. In order to take any side effects of these methods into account, any operation that may cause coercions must be wrapped. The end result of this is that the inlined code ends up emulating the interpreter, leaving no advantages to the shadow-variable approach. An alternative approach is to implement the monitor itself in JavaScript [27] and perform inlining by inlining the monitor in its entirety with target code wrapped in evaluation calls to the monitor [38].

As discussed earlier, Hedin and Sabelfeld [29] propose a dynamic monitor for a core of JavaScript, as the base for JSFlow [26], [27] to track information flow in full JavaScript. The permissiveness of the mechanism relies on upgrade instructions, whose generation can be facilitated by testing [12].

Chandra and Franz [13] present a hybrid analysis for Java bytecode. The bulk of the analysis is static, with inlined dynamic checks against a security policy that might evolve during runtime.

Vogt et al. [53] modify the source code of the Firefox browser to include a hybrid information-flow tracker in order to prevent cross-site scripting attacks. The analysis is based on tainting combined with flow-sensitive intra-procedural dataflow analysis. Suggestions on improving the dynamic component of the analysis have been discussed [45].

Mozilla's ongoing project FlowSafe [20] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [4] at its core.

Chugh et al. [15] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage. The approach is motivated by placing heavyweight static analysis for the known code on the server, and the rest of the code is checked dynamically as it becomes known.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [18] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. The goal is to prevent malicious extensions from breaking confidentiality. Bandhakavi, et al. [7] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [32] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [15], the analysis is based on code rewriting that inlines checks for data produced from sensitive sources not to flow into public sinks. They detect a number of attacks present in popular web sites, both in custom code and in third-party libraries.

Guarnieri et al. [25] present Actarus, a static taint analysis for JavaScript. An empirical study with around 10,000 pages from popular web sites exposes vulnerabilities related to injection, cross-site scripting, and unvalidated redirects and forwards. Taint analysis focuses on explicit flows, leaving implicit flows out of scope.

Stefan et al. [51] present a library for dynamic information-flow control in Haskell using a notion of floating labels, related to the concept of program counter, to restrain the side effects of computations. Although they do not allow labels of references (cf. variables) to change, their primitives allow for the manipulation of labels that causes related problems. Their solution to this is to demand the programmer to annotate the program, which is comparable to the use of upgrades.

Hritcu et al. [30] introduce the notion of brackets (related to the `toLabel` construction of [51]) to control the pc. Brackets put restrictions on the control flow leaving the bracket, which makes them comparable to our notion of return label (and exception label [27]).

Just et al. [33] develop a hybrid analysis for a subset of JavaScript. A combination of dynamic tracking and intra-procedural static analysis allows capturing both explicit and implicit flows. However, the static analysis in this work does not treat implicit flows due to exceptions.

Bichhawat et al. [10] present an information flow analysis for JavaScript bytecode. This work shares the overall motivation with ours: to use on-the-fly static analysis to improve the permissiveness of the analysis. The setting is however rather different: the analysis is implemented as instrumented runtime system for the WebKit JavaScript engine. The implementation includes a treatment of the permissive-upgrade check.

De Groef et al. [24] present *FlowFox*, a Firefox extension based on *secure multi-execution* [17]. Multi-execution runs the original program at different security levels and synchronizes communication among them. Austin and Flanagan [6] show how secure multi-execution can be optimized by executing a single program on *faceted values* rather than executing the program multiple times.

The empirical studies above [24], [25], [32], [53] provide clear evidence that privacy and security attacks in JavaScript code are a real threat. The goal of our work is to bridge the gap between the formal and practical approaches to information flow tracking, striving for permissiveness, while maintaining soundness.

With the rationale to balance precision and performance, Kerschbaumer et al. [34] probabilistically switch between two JavaScript interpreters: a fast taint-based interpreter and a slower information-flow interpreter. The information-flow interpreter utilizes a control-flow stack in a fashion similar to Vogt et al. [53]. The evaluation includes experiments with Alexa Top pages, demonstrating how the balance precision and performance can be traded in practice.

VIII. Conclusions

Leveraging an innovative synergy of static and dynamic analysis, we have developed a value-sensitive hybrid monitor for a core of JavaScript. We achieve the best of both worlds by a dynamic monitor empowered to invoke a static component on the fly. This enables us to achieve a sound yet permissive enforcement. We have established formal soundness results with respect to the security policy of noninterference. We have demonstrated permissiveness by proving that we subsume the precision of purely static analysis and by presenting a collection of common programming patterns that indicate that our mechanism provides more permissiveness than dynamic mechanisms in practice.

Future work is centered on the implementation of the extension to the full the ECMA-262 (v.5) standard [19]. Empirical evaluation is one of the important future goals, with a number of design choices to explore for better performance. Our path from theory to practice mirrors the paths taken by the related efforts previously: from theory of dynamic information flow control for JavaScript [29] to practice [27], and from theory of secure multi-execution [17] to practice for JavaScript [24].

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

References

[1] Interactive interpreter, its source code and full version of this paper available at http://jsflow.net/hybrid.

[2] Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D. Termination-insensitive noninterference leaks more than just a bit. In *Proc. ESORICS* (Oct. 2008), vol. 5283 of *LNCS*, Springer-Verlag, pp. 333–348.

[3] Askarov, A., and Sabelfeld, A. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE CSF* (July 2009).

[4] Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *Proc. ACM PLAS* (June 2009).

[5] Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *Proc. ACM PLAS* (June 2010).

[6] Austin, T. H., and Flanagan, C. Multiple facets for dynamic information flow. In *POPL* (Jan. 2012).

[7] BANDHAKAVI, S., TIKU, N., PITTMAN, W., KING, S. T., MADHUSUDAN, P., AND WINSLETT, M. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM* (2011).

[8] BARNES, J., AND BARNES, J. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] BELLO, L., AND BONELLI, E. On-the-fly inlining of dynamic dependency monitors for secure information flow. In *Proc. of FAST'11* (2011).

[10] BICHHAWAT, A., RAJANI, V., GARG, D., AND HAMMER, C. Information flow control in webkit's javascript bytecode. In *POST* (2014).

[11] BIELOVA, N. Survey on Javascript security policies and their enforcement mechanisms in a web browser. *J. Log. Algebr. Program.* (2013).

[12] BIRGISSON, A., HEDIN, D., AND SABELFELD, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS* (2012).

[13] CHANDRA, D., AND FRANZ, M. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC* (2007).

[14] CHUDNOV, A., AND NAUMANN, D. A. Information flow monitor inlining. In *Proc. of CSF'10* (2010).

[15] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for JavasCript. In *PLDI* (2009).

[16] DENNING, D. E., AND DENNING, P. J. Certification of programs for secure information flow. *Comm. of the ACM* (July 1977).

[17] DEVRIESE, D., AND PIESSENS, F. Non-interference through secure multi-execution. In *SSP* (May 2010).

[18] DHAWAN, M., AND GANAPATHY, V. Analyzing information flow in javascript-based browser extensions. In *ACSAC* (2009).

[19] ECMA INTERNATIONAL. ECMAScript Language Specification, 2009. Version 5.

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

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

[22] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *Proc. IEEE SP* (Apr. 1982).

[23] GOOGLE. Google Analytics. http://www.google.com/analytics/, 2014.

[24] GROEF, W. D., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, F. Flowfox: a web browser with flexible and precise information flow control. In *CCS* (2012).

[25] GUARNIERI, S., PISTOIA, M., TRIPP, O., DOLBY, J., TEILHET, S., AND BERG, R. Saving the world wide web from vulnerable JavaScript. In *ISSTA* (2011).

[26] HEDIN, D., BELLO, L., BIRGISSON, A., AND SABELFELD, A. JSFlow. Software release. Located at http://chalmerslbs.bitbucket.org/jsflow, Sept. 2013.

[27] HEDIN, D., BIRGISSON, A., BELLO, L., AND SABELFELD, A. JSFlow: Tracking information flow in javascript and its APIs. In *SAC* (2014).

[28] HEDIN, D., AND SABELFELD, A. A perspective on information-flow control. *Proc. of the 2011 Marktoberdorf Summer School. IOS Press* (2011).

[29] HEDIN, D., AND SABELFELD, A. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF* (June 2012).

[30] HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. All your ifcexception are belong to us. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 3–17.

[31] HUNT, S., AND SANDS, D. On flow-sensitive security types. In *POPL* (2006).

[32] JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *CCS* (2010).

[33] JUST, S., CLEARY, A., SHIRLEY, B., AND HAMMER, C. Information Flow Analysis for JavaScript. In *Proc. ACM PLASTIC* (USA, 2011), ACM.

[34] KERSCHBAUMER, C., HENNIGAN, E., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. CrowdFlow: Efficient Information Flow Security. In *ISC* (2013).

[35] LE GUERNIC, G. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE CSF* (July 2007).

[36] LE GUERNIC, G. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[37] LE GUERNIC, G., BANERJEE, A., JENSEN, T., AND SCHMIDT, D. Automata-based confidentiality monitoring. In *Proc. ASIAN* (2006).

[38] MAGAZINIUS, J., HEDIN, D., AND SABELFELD, A. Architectures for inlining security monitors in web applications. In *ESSoS* (2014).

[39] MAGAZINIUS, J., RUSSO, A., AND SABELFELD, A. On-the-fly inlining of dynamic security monitors. *Computers & Security* (2012).

[40] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java information flow. Located at http://www.cs.cornell.edu/jif, July 2001.

[41] NIELSON, F., RIIS NIELSON, H., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[42] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *CCS* (Oct. 2012).

[43] RUDERMAN, J. The same origin policy. At http://www-archive.mozilla.org/projects/security/components/same-origin.html, Apr. 2008.

[44] RUSSO, A., AND SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE CSF* (July 2010).

[45] RUSSO, A., SABELFELD, A., AND CHUDNOV, A. Tracking information flow in dynamic tree structures. In *Proc. ESORICS* (Sept. 2009).

[46] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE J. Selected Areas in Communications* (Jan. 2003).

[47] SABELFELD, A., AND RUSSO, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. PSI* (June 2009).

[48] SANTOS, J. F., AND REZK, T. An information flow monitor-inlining compiler for securing a core of javascript. In *SEC* (2014).

[49] SHROFF, P., SMITH, S., AND THOBER, M. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE CSF* (July 2007).

[50] SIMONET, V. The Flow Caml system. At http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

[51] STEFAN, D., RUSSO, A., MITCHELL, J. C., AND MAZIÈRES, D. Flexible dynamic information flow control in haskell. In *Haskell* (2011).

[52] TALY, A., ERLINGSSON, U., MILLER, M., MITCHELL, J., AND NAGRA, J. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE SP* (2011).

[53] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. NDSS* (Feb. 2007).

[54] VOLPANO, D. Safety versus secrecy. In *Proc. Symp. on Static Analysis* (1999).

[55] VOLPANO, D., SMITH, G., AND IRVINE, C. A sound type system for secure flow analysis. *J. Computer Security* (1996).

[56] W3C. Content security policy 1.0. At http://www.w3.org/TR/CSP/, Nov. 2012.

[57] W3C. Cross-origin resource sharing. At http://www.w3.org/TR/CSP/, Jan. 2014.

[58] ZDANCEWIC, S. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.