Value Sensitivity and Observable Abstract Values for Information Flow Control

Luciano Bello[1], Daniel Hedin[1,2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] Mälardalen University

Abstract. Much progress has recently been made on information flow control, enabling the enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as JavaScript, Java, Caml, and Ada that enforce versatile security policies. However, a roadblock on the way to wider adoption of these tools has been their limited permissiveness (high number of false positives). Flow-, context-, and object-sensitive techniques have been suggested to improve the precision of static information flow control and dynamic monitors have been explored to leverage the knowledge about the current run for precision.
This paper explores *value sensitivity* to boost the permissiveness of information flow control. We show that both dynamic and hybrid information flow mechanisms benefit from value sensitivity. Further, we introduce the concept of *observable abstract values* to generalize and leverage the power of value sensitivity to richer programming languages. We demonstrate the usefulness of the approach by comparing it to known disciplines for dealing with information flow in dynamic and hybrid settings.

1 Introduction

Much progress has recently been made on information flow control, enabling the enforcement of increasingly rich policies for increasingly expressive programming languages. This has resulted in tools for mainstream programming languages as FlowFox [16] and JSFlow [20] for JavaScript, Jif [26], Paragon [9] and JOANA [17] for Java, FlowCaml [30] for Caml, LIO [31] for Haskell, and SPARK Examiner [5] for Ada that enforce versatile security policies. However, a roadblock on the way to wider adoption of these tools has been their limited permissiveness i.e secure programs are falsely rejected due to over-approximations. Flow-, context-, and object-sensitive techniques [17] have been suggested to improve the precision of static information flow control, and dynamic and hybrid monitors [22, 32, 27, 20, 19] have been explored to leverage the knowledge about the current run for precision. Dynamic and hybrid techniques are particularly promising for highly dynamic languages such as JavaScript. With dynamic languages as longterm goal, we focus on fundamental principles for sound yet permissive dynamic information flow control with possible static enhancements.

In dynamic information flow control, each value is associated with a runtime *security label* representing the *security classification* of the value. These labels are propagated during computation to track the flow of information through the program. There are two basic kinds of flows: *explicit* and *implicit* [14]. The former is induced by *data flow*, e.g., when a value is copied from one location to another, while the latter is induced by *control flow*. The following example of implicit flow leaks the boolean value of h into l with no explicit flow involved: `l = false;if (h) l = true;`

Dynamic information flow control typically enforces *termination-insensitive non-interference* (TINI) [33]. Under a two-level classification into *public* and *secret* values, TINI demands that values labeled public are independent of values labeled secret in *terminating runs* of a program. Note that this demand includes the label itself, which has the effect of constraining how security labels are allowed to change during computation. This is a fundamental restriction: freely allowing labels to change allows circumventing the enforcement [27].

A common approach to securing label change is the *no secret upgrade (NSU)* restriction that forbids labels from changing under *secret control*, i.e., when the control flow is depending on secrets [2]. In the above example, NSU would stop the execution when h is `true`. This enforces TINI because in all *terminating* runs the l is untouched and hence independent of h.

Unfortunately, his limitation of *pure dynamic* information flow control often turns out to be too restrictive in practice [20], and various ways of lifting the restriction have been proposed [3, 8]. They aim to enhance the dynamic analysis with information that allows the label of *write target* to be changed before entering secret control, thus decoupling the label change from secret influence. For instance, a *hybrid* approach [22, 32, 27, 19] is to apply a static analysis on the bodies of *elevated contexts*, e.g., secret conditionals, to find all potential write targets and upgrade them before the body is executed.

This paper investigates an alternative approach that improves both pure and hybrid dynamic information flow control as well as other approaches relying on upgrading labels before elevated contexts. The approach increases the *precision* of the labeling, hence reducing the number of elevated contexts. In a pure dynamic analysis this has the effect of reducing the number of points in the program where execution is stopped with a security error, while in a hybrid approach this reduces the number of places the static analysis invoked further improving the precision by not unnecessarily upgrading write targets.

Resting on a simple core, the approach is surprisingly powerful. We call the mechanism *value sensitive*, since it considers the previous target *value* of a monitored side-effect and, if that value remains *unchanged* by the update, the security label is left untouched. Consider the program in Listing 1.1. It is safe to allow execution to continue even when h is `true` by effectively ignoring the update of l in the body of the conditional. This still satisfies TINI because all runs of the program leaves l untouched and independent of h.

Listing 1.1

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

The generalization of the idea boosts permissiveness when applied to other notions of values, e.g. the type of a variable, as exemplified on the right.

In a dynamically typed language the value of t changes from a public to a secret value, but the (dynamic) type of t remains unchanged. By tracking the type of t independently of its value (for example as $\langle value^{\sigma}, type^{\sigma'} \rangle$), it is possible to leverage value sensitivity and allow the security label of the type to remain public. Thus, l is tagged L, which is safe and more precise than under traditional monitoring.

```
t = 2^L;
t = 1^H;
l = typeof(t);
```

Similarly, if we consider a language with records, the following snippet illustrates that the field existence of a property can be observable independently. In a language with observable existence (in this case through the primitive `in`) a monitor might gain precision by labeling this feature independently of the value. The label does not need to be updated when the property assignment is run, since the existence of the property remains the same.

```
o = { p: 2^L };
o['p'] = 1^H;
l = 'p' in o;
```

The type and the existence are two examples of properties of runtime values that can be independently observed and change less often than the values. We refer to such properties as *Observable Abstract Values* (OAV). Value sensitivity can be applied to any OAVs. The synergy between these two concepts has the power to improve existing purely dynamic and hybrid information flow monitors, as well as improving existing techniques to handle advanced data types as dynamic objects. The main contributions of this paper are

- the introduction of the concept of *value sensitivity* in the setting of *observable abstract values*, realized by systematic use of *lifted maps*,
- showing how the notion of value sensitivity naturally entails the notion of *existence* and *structure* labels, frequently used in the analysis of *dynamic objects* in addition to improving the precision of previous techniques while significantly simplifying the semantics and correctness proofs.
- the application of value sensitivity to develop a novel approach to *hybrid* information flow control, where not only the underlying *dynamic* analysis but also the *static* counterpart is improved by value sensitivity.

We believe that systematic application of value sensitivity on identified observable abstract values can serve as a method when designing dynamic and hybrid information flow control mechanism for new languages and language constructs. The full version [1] of the paper contains the full details and proofs.

2 The core language $\mathcal{L}$

We illustrate the power of the approach on a number of specialized languages formulated as extensions to a small while language $\mathcal{L}$, defined as follows.

$$e ::= l \mid e \oplus e \mid x \mid x \texttt{=} e \qquad s ::= \texttt{if(}e\texttt{)}\{s\}\{s\} \mid \texttt{while(}e\texttt{)}\{s\} \mid s\texttt{;}s \mid \texttt{skip} \mid e$$

The expressions consist of literal values l, binary operators abstractly represented by $\oplus$, variables and variable assignments. The statements are built up by

$$\textsc{Assign}\frac{\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\bot}} \dot{v}]\downarrow_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x \,\texttt{=}\, e\rangle \to_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

$$\textsc{If}\frac{\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, v^{\sigma}\rangle \quad \langle \mathcal{S}_2, s_v\rangle \to_{pc\sqcup\sigma} \mathcal{S}_3}{\langle \mathcal{S}_1, \texttt{if}\,(e)\,\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}\rangle \to_{pc} \mathcal{S}_3} \qquad \textsc{Var}\frac{\mathcal{S}_{\texttt{undef}^{\bot}}(x) = \dot{v}}{\langle \mathcal{S}, x\rangle \to_{pc} \langle \mathcal{S}, \dot{v}\rangle}$$

Fig. 1: Partial $\mathcal{L}$ semantics

conditional branches, while loops, sequencing and skip, with expressions lifted into the category of statements.

$$\mathcal{S} : \textit{string} \to \textit{LabeledValue} \quad v ::= \textit{bool} \,\big|\, \textit{integer} \,\big|\, \textit{string} \,\big|\, \texttt{undef}$$
$$\dot{v} \in \textit{LabeledValue} ::= v^{\sigma} \quad C ::= \langle \mathcal{S}, \dot{v}\rangle \,\big|\, \mathcal{S} \quad pc, \sigma, \omega \in \textit{Label}$$

The semantics of the core language is a standard dynamic monitor. The primitive values are booleans, integers, strings and the distinguished `undef` value returned when reading a variable that has not been initialized. The values are labeled with security labels drawn from a lattice of security labels, *Label*. Let $\bot \in \textit{Label}$ denote the least element. Unless indicated otherwise, in the examples, a two-point lattice $L \sqsubseteq H$ is used, representing *low* for public information and *high* for secret. The label operator $\sqcup$ notates the least-upper-bound in the lattice.

The semantics is a big-step semantics of the form $\langle \mathcal{S}, s\rangle \to_{pc} C$ read as: the statement s executing under the label of the program counter *pc* and initial state $\mathcal{S}$ results in the configuration C. The states are *partial maps* from variable names to labeled values and the configurations are either states or pairs of states and values.

The main elements of the semantic are described in Figure 1, with the remaining rules in the [1] for space reasons. The selected rules illustrate the interplay between conditionals, the *pc* and assignment. The IF rule elevates the *pc* to the label of the guard and evaluates the branch taken under the elevated *pc*. The VAR rule and the ASSIGN rule, for variable look up and side effects, use operations on the *lifted partial map*, $\mathcal{S}_{\texttt{undef}^{\bot}}$, to read and write to variables respectively. In the latter case, this is where the *pc* constrains the side effects.

Lifted partial maps provide a generic way to safely interact with partial maps with labeled codomains. For example, as shown in Figure 1, a lifted partial map is used to interact with the variable environment. In general, lifted partial maps are very versatile and in Section 3 will be used to model a variety of aspects.

A lifted partial map is a partial map with a default value. For a partial map $\mathcal{M} : X \to Y$, the map $\mathcal{M}_{\varDelta} : X \to Y \cup \varDelta$ is the lifted map with default value $\varDelta$, where $\mathcal{M}_{\varDelta}(x) = \begin{cases} \mathcal{M}(x) & x \in \textit{dom}(\mathcal{M}) \\ \varDelta & \text{otherwise} \end{cases}$ This defines the reading operation. For writing, $\mathcal{M}[x \xleftarrow{\varDelta} \dot{v}]\downarrow_{pc} \mathcal{M}'$ denotes that x is safely updated with the value $\dot{v}$ in the partial map $\mathcal{M}$, resulting in the new partial map $\mathcal{M}'$. Formally, the MUPDATE rule governs this side-effect as follows:

To update the element x of a lifted partial map with a labeled value $\dot{v}$, the current value of x needs to be fetched.

$$\textsc{MUpdate}\ \frac{\mathcal{M}_{\Delta}(x) = w^{\omega} \quad pc \sqsubseteq \omega}{\mathcal{M}[x \overset{\Delta}{\leftarrow} \dot{v}] \downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

To block implicit leaks, the label of this value, ω, has to be above the level of the context, *pc*. In terms of the variable environment above, if a variable holds a low value, it cannot be updated in a high context. If the update is allowed, the label of the new value is lifted to the *pc* ($\dot{v}^{pc}$) before being stored in x. This implements the standard NSU restriction.

However, there is a situation where this restriction can be relaxed: when the the variable to update already holds the value to write, i.e., when the side-effect is not observable. In this case, the update can be safely ignored rather than causing a security error, even if the target of the side-effect is not above the level of the context.

$$\textsc{MUpdate-VS}\ \frac{\begin{array}{c}\mathcal{M}_{\Delta}(x) = w^{\omega} \\ pc \not\sqsubseteq \omega \quad w = v\end{array}}{\mathcal{M}[x \overset{\Delta}{\leftarrow} \dot{v}] \downarrow_{pc} \mathcal{M}}$$

The MUpdate-VS rule extends the permissiveness of the monitor in cases where $pc \not\sqsubseteq \omega$, like in Listing 1.1. Intuitively, the assignment statement does not break the NSU invariant and it is safe to allow it. We call an enforcement that takes the previous value of the write target into account *value-sensitive*.

Note that, in the semantics, security errors are not explicitly modeled - rather they are manifested as the failure of the semantics to progress. In a semantics with only the MUpdate rule, any update that does not satisfy the demands will cause execution to stop. The addition of MUpdate-VS however allows the special case, where the value does not change, to progress.

3 Observable Abstract Values

The notion of value sensitivity naturally scales from values to other properties of the semantics. Any property that can act as mutable state, i.e., that can be read and written, is a potential candidate. In the case where the property changes less frequently than the value, such a modeling may increase the precision. In particular, assuming that the property is modeled with a security label of its own, the NSU label check can be omitted when an idempotent operation, with respect to the property, is performed. We refer to such properties as *Observable Abstract Values* (OAV). Consider the following examples of OAVs:

– **Dynamic types** It is common that the value held by a variable is secret, while its type is not. In addition, values of variables change more frequently than types which means that most updates of variables do not change the type.
– **Property existence** The existence of properties in records or objects can be observed independently of their value. Changing a value in a property does not affect its existence.
– **List or array length** Related to property existence, the length of a list or array is independent of the values. Mutating the list or the array without adding or deleting values does not affect the length.

– **Graph/tree structure** More generally, not only the number of nodes in a data structure, but any observable structural characteristic can be modeled as OAVs, such as tree height.
– **Security labels** Sometimes [9, 10] the labels on the values are observable. Since they change less often than the value themselves, they can be modeled as OAVs.

Different OAVs are not necessarily independent. In the same way an OAV is an abstraction of a value, it is possible to find OAVs that are natural abstractions of other OAVs. Such partial order is of interest both from an implementation and proof perspective. For space reasons we refer the reader to the full version [1] of the paper for more information.

The rest of this section explains the first two examples above as extensions of the core language $\mathcal{L}$. The extension with dynamic types $\mathcal{L}_t$ is detailed in Section 3.1, and the extension with records modeling existence and structure $\mathcal{L}_r$ is detailed in Section 3.2. The two latter extensions illustrate that the approach subsumes and improves previous handling of records [18].

3.1 Dynamic types $\mathcal{L}_t$

Independent labeling of OAVs allows for increased precision when combined with value sensitivity. To illustrate this point, consider the example in Listing 1.2 where the types are independently observable from the values themselves, via the primitive `typeof()`. Assuming that the value of t is initially secret while the type is not, the example in the listing illustrates how the value of t is made dependent on h while the type remains independent.

Listing 1.2

```
t = ⟨1^H,int^L⟩;
if (h) {t = 2;} else {t = 3;}
l = typeof(t);
```

The precision gain is significant for, e.g., JavaScript. A common defensive programming pattern for JavaScript library code is to probe for the presence of needed functionality in order to fail gracefully in case it is absent. Consider, for instance, a library that interacts with `document.cookie`. Even if all browsers support this particular property, it is dangerous for a library to assume that it is present, since the library might be loaded in, e.g, a sandbox that removes parts of the API. For this reason it is very common for libraries to employ the defensive pattern shown in Listing 1.3, where the dots represent the entire library code. While the value of `document.cookie` is secret its presence is not. If no distinction between the type of a value and its actual value is made this would cause the entire library to execute under secret control.

Listing 1.3

```
if (typeof document.cookie
    !== "undefined") { ... }
```

To illustrate this scenario, we extend $\mathcal{L}$ with dynamic types and a `typeof()` operation that given an expression returns a string representing the type of the expression:

$$e ::= (\cdots \text{as in } \mathcal{L}) \bigm| \texttt{typeof}(e) \quad s ::= (\cdots \text{as in } \mathcal{L})$$

$$\textsc{Assign}_t \frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle\langle \mathcal{V}_2, \mathcal{T}_2\rangle, \langle \dot{v}, \dot{t}\rangle\rangle \quad \mathcal{V}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}]{\downarrow_{pc}}\, \mathcal{V}_3 \quad \mathcal{T}_2[x \xleftarrow{undef^{\perp}} \dot{t}]{\downarrow_{pc}}\, \mathcal{T}_3}{\langle \mathcal{S}_1, x\,\texttt{=}\,e\rangle \rightarrow_{pc} \langle\langle \mathcal{V}_3, \mathcal{T}_3\rangle, \langle \dot{v}, \dot{t}\rangle\rangle}$$

$$\textsc{Typeof} \frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t}\rangle\rangle}{\langle \mathcal{S}_1, \texttt{typeof(}e\texttt{)}\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle string(\dot{t}), str^{\perp}\rangle\rangle}$$

$$\textsc{Var}_t \frac{\mathcal{V}_{\texttt{undef}^{\perp}}(x) = \dot{v} \quad \mathcal{T}_{undef^{\perp}}(x) = \dot{t}}{\langle\langle \mathcal{V}, \mathcal{T}\rangle, x\rangle \rightarrow_{pc} \langle\langle \mathcal{V}, \mathcal{T}\rangle, \langle \dot{v}, \dot{t}\rangle\rangle}$$

Fig. 2: Partial $\mathcal{L}_t$ semantics

The semantics is changed to accommodate dynamically typed values. In particular typed values are pairs of a security labeled value, and a security labeled dynamic type. Additionally, the state $\mathcal{S}$ is extended to a tuple holding the value context $\mathcal{V}$ and the type context $\mathcal{T}$.

$$\mathcal{V} : string \rightarrow LabeledValue \quad \mathcal{T} : string \rightarrow LabeledType$$
$$TypedValue ::= \langle \dot{v}, \dot{t}\rangle \quad t \in Type ::= bool \,\big|\, int \,\big|\, str \,\big|\, undef \quad \mathcal{S} \in State ::= \langle \mathcal{V}, \mathcal{T}\rangle$$

A consequence of the extension with dynamic types is that the semantic rules must be changed to operate on typed values. Figure 2 contains the most interesting rules - the remaining rules can be found in the full version of this paper [1].

The `typeof()` operator (Typeof) returns a string representation of the type of the given expression. The string inherits the security label of the type of the expression, whereas the type of the result is always *str* and hence labeled $\perp$.

Further, the rules for variable assignment ($\textsc{Assign}_t$) and variable look-up ($\textsc{Var}_t$) require special attention. Notice that, for both maps $\mathcal{V}$ and $\mathcal{T}$, the default lookup value is undefined: $\texttt{undef}^{\perp}$ and $undef^{\perp}$ respectively. These maps are independently updated through $\textsc{Assign}_t$, which calls MUpdate and MUpdate-VS accordingly. Variable look up is the reverse process: the type and value are fetched independently from their respective maps.

If we return to the example in Listing 1.2, the value of t is updated but not its type. Therefore, under a value-sensitive discipline, the execution is safe and `l` will be assigned to $\langle "int"^{L}, str^{L}\rangle$ at the end of the execution.

Distinguishing between the type of a value and its actual value in combination with value sensitivity is an important increase in precision for practical analyses. It allows the execution of the example of wild JavaScript from Listing 1.3, since `typeof document.cookie` returns $\langle "str"^{\perp}, str^{\perp}\rangle$, which makes the result of the guarding expression public.

3.2 Records and observable property existence $\mathcal{L}_r$

Previous work on information flow control for complex languages has used the idea of tracking the existence of elements in structures like objects with an

$$\textsc{RecAssign}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e_1\rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \langle \mathcal{S}_2, e_2\rangle \to_{pc} \langle \mathcal{S}_3, \dot{v}\rangle \\ \mathcal{S}_3(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \quad \sigma = pc \sqcup \sigma_f \\ \mathcal{V}_1[f \xleftarrow{\texttt{undef}^{\varsigma}} \dot{v}]{\downarrow_\sigma}\ \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\texttt{false}^{\varsigma}} \texttt{true}^{\perp}]{\downarrow_\sigma}\ \mathcal{E}_2\end{array}}{\langle \mathcal{S}_1, x\texttt{[}e_1\texttt{]}\,\texttt{=}\,e_2\rangle \to_{pc} \langle \mathcal{S}_3[x \to \langle \mathcal{V}_2, \mathcal{E}_2\rangle_{\varsigma}^{\sigma_x}], \dot{v}\rangle}$$

$$\textsc{Proj}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \\ \mathcal{V}_{\texttt{undef}^{\varsigma}}(f) = \dot{v} \quad \sigma = \sigma_x \sqcup \sigma_f\end{array}}{\langle \mathcal{S}_1, x\texttt{[}e\texttt{]}\rangle \to_{pc} \langle \mathcal{S}_2, \dot{v}^{\sigma}\rangle}$$

$$\textsc{In}\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \\ \mathcal{E}_{\texttt{false}^{\varsigma}}(f) = \dot{v} \quad \sigma = \sigma_x \sqcup \sigma_f\end{array}}{\langle \mathcal{S}_1, e\,\texttt{in}\,x\rangle \to_{pc} \langle \mathcal{S}_2, \dot{v}^{\sigma}\rangle}$$

Fig. 3: $\mathcal{L}_r$ semantics extension over $\mathcal{L}$

independent existence label [28, 18, 24]. In this section, we show that the notion of OAVs and the use of lifted partial maps are able to naturally express previous models while significantly simplifying the rules. Further, systematic application of those concepts allows us to improve previous models — in particular for property deletion.

Treating the property existence separately increases the permissiveness of the monitor. Consider, for instance, the example in Listing 1.4. After execution, the value of property x depends on h but not its existence. Since the existence changes less often and is observable via the operator `in`, it can be seen as an OAV (of the record).

Listing 1.4

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

In order to reason about existence as an OAV, we create $\mathcal{L}_r$ by extending $\mathcal{L}$ with record literals, property projection, property update and an `in` operator that makes it possible to check if a property is present in a record.

$$e ::= (\cdots \text{as in } \mathcal{L}) \mid \{\overline{e : e}\} \mid x\texttt{[}e\texttt{]} \mid x\texttt{[}e\texttt{]}\,\texttt{=}\,e \mid e\,\texttt{in}\,x \qquad s ::= (\cdots \mathcal{L})$$

The records are implemented as tuples of maps $\langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}$ decorated with a *structure security label* ς.

$$\mathcal{V} : \mathit{string} \to \mathit{LabeledValue} \quad \mathcal{E} : \mathit{string} \to \mathit{LabeledBool}$$
$$\mathcal{S} : \mathit{string} \to \mathit{LabeledValue} \quad v ::= r \mid (\cdots \text{as in } \mathcal{L}) \quad r ::= \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}$$

The first map, $\mathcal{V}$, stores the labeled values of the properties of the record, and the second map $\mathcal{E}$ stores the presence (existence) of the properties as a labeled boolean. As in previous work, the interpretation is that present properties carry their own existence label while inexistent properties are modeled by the structure label. As we will see below, the structure label is tightly connected to (the label of) the default value of $\mathcal{V}$ and $\mathcal{E}$. For clarity of exposition we let the records be values rather than entities on a heap.

The semantics of property projection, assignment, and existence query are detailed in Figure 3. Property update (RecAssign) allows for the update of a property in a record stored in a variable and the projection rule (Proj) reads a property by querying only the map $\mathcal{V}$. There are a number of interesting properties of these two rules. For RecAssign note the uniform treatment of values and existence and how, in contrast to previous work, this simplifies the semantics to only one rule. Further, note how the structure label is used as the label of the default value in both rules and how this interacts with the rules for lifted partial maps.

Listing 1.5

```
o={ e_L: 0^L,
f_L: 1^M, g_H: 2^H}_H;
if ( m^M ) {
   o['e'] = 0;
   o['h'] = 0;
   o['f'] = 0;
   o['g'] = 0;
}
```

Consider Listing 1.5 in a $L \sqsubset M \sqsubset H$ security lattice to illustrate the logic behind this monitor. In this example, the subindex label in the key of the record denotes the existence label for that property. When the true branch is taken, the assignment `o['e']=0` (on line 4) is ignored, since MUpdate-VS is applied. Although the context is higher than the label of the value and its existence, no label change will occur.

The second assignment (`o['h']=0`, on line 5) extends the record. This side effect demands that the structure label of the record is not below M. The demand stems from the MUpdate rule via the label of the default value and initiated by the update of the existence map from false to true. Since the value changes only MUpdate is applicable, which places the demand that the label of the previous value (the structure label) is above the label of the control. The new value is tainted with the label of the control, which in this case leads to an existence label of M, resulting in `{ ..., h`$_M$`:0`^{M}`}`$_H$.

To contrast, consider the next property update (`o['f']=0`, on line 6), which writes to a previously existing property under M control. In this case no demands will be placed on the structure label, since neither of the maps will trigger use of the default value. The previous existence label is below M, but this does not trigger NSU since the value of the existence does not change, which makes the MUpdate-VS rule is applicable. This also means that the existence label is untouched and the result after execution is `{ ..., f`$_L$` : 0`^{M}`, ...}`$_H$.

Finally (`o['g']=0`, on line 7), the previous existence and value labels are both above M, and the MUpdate rule is applicable. This will have the effect of *lowering* both the existence and value label to then current context in accordance with flow-sensitivity. The result after execution is `{ ..., g`$_M$` : 0`^{M}`, ... }`$_H$

It is worth noting that the above example can be easily recast to illustrate update using a secret property name, since the pc and the security label of the property name form the security context, σ, of the writes in RecAssign.

With respect to reading, the existence label is not taken into account unless reading a non-existent property, in which case the structure of the record is used via the default value. Analogously, the rule In checks for property existence in a record by performing the same action on the $\mathcal{E}$ map. This illustrates that the lifted maps provide a natural model for existence tracking. The existence map provides all the presence/absence information of a value in a particular property. This generalization, in combination with value sensitivity, both simplifies previous work and increases the precision of the tracking. In particular, as shown in the full version [1] of the paper, this is true when property deletion is considered.

$$\textsc{S-If}\ \frac{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow_{pc} \mathcal{S}_t \quad \langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow_{pc} \mathcal{S}_f}{\langle \mathcal{S}_1, \texttt{if}\,(e)\,\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}\rangle \Rightarrow_{pc} \mathcal{S}_t \sqcup \mathcal{S}_f}$$

$$\textsc{S-Assign}\ \frac{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}] \Downarrow_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x \texttt{=} e\rangle \Rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

$$\textsc{If}_h\ \frac{\begin{array}{c}\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^{\sigma}\rangle \quad \langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_t \\ \langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_f \quad \langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_v\rangle \rightarrow_{pc\sqcup\sigma} \mathcal{S}_3\end{array}}{\langle \mathcal{S}_1, \texttt{if}\,(e)\,\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}\rangle \rightarrow_{pc} \mathcal{S}_3}$$

Fig. 4: Partial hybrid semantics

4 Hybrid monitors $\mathcal{L}_h$

In the quest of more permissive dynamic information flow monitors, *hybrid monitors* have been developed. Some perform static *pre-analyzes*, i.e., before the execution [13, 21, 25], or code inlining [12, 6, 23, 29]. In other cases, the static analysis is triggered at runtime by the monitor [22, 32, 27, 19]. A value sensitivity criterion can be applied in the static analysis of this second group. This means that fewer potential write targets need to be considered by the static part of these monitors.

Consider, for instance Listing 1.1, where a normal (i.e., value *insensitive*) hybrid monitor would elevate the label of l to the label of h before evaluating the branch. A value-sensitive hybrid analysis, on the other hand, is able to avoid the elevation, since the value of l can be seen not to change in the assignment.

To illustrate how a hybrid value-sensitive monitor might work consider the following hybrid semantics for the core language. Syntactically, $\mathcal{L}_h$ is identical to $\mathcal{L}$ but, similar to [22] and [19], a static analysis is performed when a branching is reached.

Consider the rule for conditionals ($\textsc{If}_h$) that applies a static analysis on the body of the conditional in order to update any variables that are potential write targets. In particular, assignments will be statically executed (S-Assign), which elevates the target to the current context using static versions of MUpdate and MUpdate-VS. This means that the NSU check of MUpdate no longer needs to be performed — the static part of the analysis guarantees that all variables are updated before execution. The static update and new dynamic update rules are formulated as follows.

$$\textsc{S-MUpdate}\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w \neq v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \Downarrow_{\sigma} \mathcal{M}[x \mapsto \dot{w}^{\sigma}]} \qquad \textsc{MUpdate}_h\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w \neq v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

$$\textsc{S-MUpdateVS}\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \Downarrow_{\sigma} \mathcal{M}} \qquad \textsc{MUpdate-VS}_h\ \frac{\mathcal{M}_{\Delta}(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}] \downarrow_{pc} \mathcal{M}}$$

The value sensitivity of the static rules is manifested in the S-MUpdateVS rule. In case the new value is equal to the value of the write target, no label elevation is performed, which increases the permissiveness of the hybrid monitor in the way illustrated in Listing 1.1. Note the similarity between the static and the dynamic rules. In case it can be statically determined that the value does not change we know that MUpdate-VS$_h$ will be run at execution time and vice versa for MUpdate$_h$. This allows for the increase in permissiveness while still guaranteeing soundness. Naturally, this development scales to general OAVs under hybrid monitors.

5 Permissiveness

Value-sensitive monitors are strictly more permissive than their value-insensitive counterparts with respect to *termination insensitive non-interference* (TINI). This means that value-sensitive discipline accepts more safe programs without allowing insecure programs to be executed.

For space reasons, the soundness proof can be found in the full version of this paper [1].

In this section we compare the value sensitive languages $\mathcal{L}$, $\mathcal{L}_{rd}$ and $\mathcal{L}_h$ to the value-insensitive counterparts. In particular $\mathcal{L}$ is comparable to the Austin and Flanagan NSU discipline [2], $\mathcal{L}_{rd}$ is compared to the record subset of JSFlow [20] and $\mathcal{L}_h$ is compared to the Le Guernic et al.'s hybrid monitor [22].

5.1 Comparison with Austin & Flanagan's NSU [2]

The comparison with non-sensitive upgrade is relatively straight forward, since $\mathcal{L}$ is essentially the NSU monitor of [2] with one additional value-sensitive rule, MUpdate-VS.

Let $\dashrightarrow$ denote reductions in the insensitive monitor obtained by removing MUpdate-VS from $\mathcal{L}$. To show permissiveness we will prove that every reduction $\dashrightarrow$ can be followed by a reduction $\rightarrow$.

Theorem 1 (value-sensitive NSU is strictly more permissive than value-insensitive NSU).

$$\forall s \in \mathcal{L} \,.\, \langle \mathcal{S}_1, s\rangle \dashrightarrow_{pc} \mathcal{S}_2 \Rightarrow \langle \mathcal{S}_1, s\rangle \rightarrow_{pc} \mathcal{S}_2 \wedge$$
$$\exists s \in \mathcal{L} \,.\, \langle \mathcal{S}_1, s\rangle \rightarrow_{pc} \mathcal{S}_2 \not\Rightarrow \langle \mathcal{S}_1, s\rangle \dashrightarrow_{pc} \mathcal{S}_2$$

Proof. $\Rightarrow$: By contradiction, using that $\dashrightarrow$ is a strict subset of $\rightarrow$. For space reasons the proof can be found in the full version of this paper [1]. $\not\Rightarrow$: The program in Listing 1.1 proves the claim, since it is successfully executed by $\rightarrow$ but not by $\dashrightarrow$.

5.2 Comparison with JSFlow [20]

Hedin et al. [20] present JSFlow, a sound purely-dynamic monitor for JavaScript. JSFlow tracks property existence and object structure for dynamic objects with property addition and deletion. The objects are represented as $\{\overline{x \xrightarrow{\epsilon} p^{\sigma}}\}_{\varsigma}$, i.e., objects are maps from properties, x, to labeled values, p^{σ}, with properties carrying existence labels, ϵ, and objects structure labels, ς.

Consider the example in Listing 1.6 up to line 3, where the property x is added under secret control. This places the demand that the structure of o is below the pc. In $\mathcal{L}_{rd}$, this demand stems from the MUPDATE rule via the label of the default value and is initiated by the update of the existence map from false to true. For $\mathcal{L}_{rd}$ the resulting object is $\langle\{\texttt{x} \to \texttt{0}^H\}, \{\texttt{x} \to \texttt{true}^H\}\rangle_H$, while for JSFlow the resulting object would be $\{\texttt{x} \xrightarrow{H} \texttt{0}^H\}_H$.

Listing 1.6

```
0
1  o={}_H
2  if (h^H) {
3    o['x']=0;
4  }
5  delete o['x'];
6  l = 'x' in o;
```

If we proceed with the execution, the deletion on line 5 is under public context, which illustrates the main semantic difference between $\mathcal{L}_{rd}$ and JSFlow. In the former, deletion under public control will have the effect of *lowering* the value and existence labels to the current context, which results in $\langle\{\texttt{x} \to \texttt{undef}^L\}, \{\texttt{x} \to \texttt{false}^L\}\rangle_H$. In the latter, property absence is not explicitly tracked and deleting a property simply removes it from the map resulting in $\{\}_H$. Therefore, at line 6, $\mathcal{L}_{rd}$ is able to use that the absence of x is independent of secrets, while JSFlow will taint l with H based on the structure level. In this way, $\mathcal{L}_{rd}$ both simplifies the rules of previous work and increases the precision of the tracking.

5.3 Comparison with Le Guernic et al.'s hybrid monitor [22]

The hybrid monitor presented by Le Guernic et al. [22] is similar to $\mathcal{L}_h$. In both cases, a static analysis is triggered at the branching point to counter the inherent limitation of purely-dynamic monitors: that they only analyze one trace of the execution.

In the case of Le Guernic et al., the static component of their monitor collects the left-hand side of the assignments in the both sides of branches. Once these variables are gathered their labels are upgraded to the label of the branching guard. Intuitively, the targets of assignments in branch bodies depend on the guard, but as, e.g., Listing 1.1 shows this method is an over-approximation. Such over-approximations lower the precision of the enforcement, and might, in particular, when the monitor tracks OAVs rather than regular values, jeopardize the practicability of the enforcement.

The hybrid monitor $\mathcal{L}_h$ subsumes the monitor by Le Guernic et al. (see [1]). All variable side-effects taken into account by Le Guernic et al. are also considered by the static part of $\mathcal{L}_h$ via the rule for static assignment, S-ASSIGN. More precisely, S-ASSIGN updates the labels of the variables by applying either S-MUPDATE or S-MUPDATEVS depending on the previous value. The case when all variables are upgraded by S-MUPDATE to the level of the guard (σ in the rules of Figure 4) corresponds to monitor by Le Guernic et al.

6 Related Work

This paper takes a step forward to improve the permissiveness of dynamic and hybrid information flow control. We discuss related work, including work that can be recast or extended in terms of value sensitivity and OAVs.

Permissiveness Russo and Sabelfeld [27] show that flow-sensitive dynamic information flow control cannot be more permissive than static analyses. This limitation carries over to value-sensitive dynamic information flow analyses.

Austin and Flanagan extend the permissiveness of the NSU enforcement with *permissive upgrades* [3]. In this approach, the variables assigned under high context are tagged as *partially-leaked* and cannot be used for future branching. Bichhawat et al. [7] generalize this approach to a multi-level lattice. Value sensitivity can be applied to permissive upgrades (including the generalization) with benefits for the precision.

Hybrid approaches are a common way to boost the permissiveness of enforcements. There are several approaches to hybrid enforcement: inlining monitors [12, 6, 23, 29], selective tracking [13, 25], and the application of a static analysis at branch points [22, 32, 27, 19]. Value sensitivity is particularly suitable for the latter to reduce the number of upgrades and increase precision (cf. Section 4).

In relation to OAVs Some enforcements track other more abstract properties in addition to standard values. These properties are typically equipped with a dedicated security label, which makes them fit into our notion of OAV.

Buiras et al. [10] extend LIO [31] to handle flow-sensitivity. Their *labelOf* function allows them to observe the label of values. To protect from leaks through observable labels, their monitor implements a *label on the label*, which means that the label itself can be seen as an OAV.

Almeida Matos et al. [24] present a purely dynamic information flow monitor for DOM-like tree structures. By including references and live collections, they get closer to the real DOM specification but are forced to track structural aspects of the tree, like the position of the nodes. Since the attacker can observe changes in the DOM through live collections and, in order to avoid over-approximations, they label several aspects of the node: the node itself, the value stored in it, the position in the forest, and its structure. These aspects are OAVs, since some of the operations only affect a subset of their labels. A value-sensitive version of this monitor might not be trivial given its complexity, but the effort would result in increased precision.

In relation to value-sensitivity The hybrid JavaScript monitor designed by Just et al. [21] only alters the structure of objects and arrays when properties or elements are inserted or removed. Similarly, Hedin et al. [19, 20] track the presence and absence of properties and elements in objects and arrays changing the associated labels on insertions or deletions. Both approaches can be understood in terms of value-sensitivity. Indeed, in this paper we show how to improve the latter by systematic modeling using OAVs in combination with value-sensitivity.

Secure multi-execution [11, 15] is naturally value-sensitive. It runs the same program multiple times restricting the input based on its confidentiality level. In this way, the secret input is defaulted in the low execution, thus entirely decoupling the low execution from the secret input. Austin and Flanagan [4] present *faceted values*: values that, depending of the level of the observer, can *return* differently. Faceted values provide an efficient way of simulating the multiple executions of secure multi-execution in a single execution.

7 Conclusion

We have investigated the concept of value sensitivity and introduced the key concept of observable abstract values, which together enable increased permissiveness for information flow control. The identification of observable abstract values opens up opportunities for value-sensitive analysis, in particular in richer languages. The reason for this is that the values of abstract properties typically change less frequently than the values they abstract. In such cases, value-sensitivity allows the security label corresponding to the abstract property to remain unchanged.

We have shown that this approach is applicable to both purely dynamic monitors, where we reduce blocking due to false positives, and to hybrid analysis, where we reduce over-approximation.

Being general and powerful concepts, value sensitivity and observable abstract values have potential to serve as a basis for improving state-of-the-art information flow control systems. Incorporating them into the JSFlow tool [20] is already in the workings.

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

References

1. Full version at http://chalmerslbs.bitbucket.org/valsens/fullversion.pdf
2. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proc. ACM PLAS (2009)
3. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: PLAS (2010)
4. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: PoPL (2012)
5. Barnes, J., Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security (2003)
6. Bello, L., Bonelli, E.: On-the-fly inlining of dynamic dependency monitors for secure information flow. In: FAST (2011)
7. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Generalizing permissive-upgrade in dynamic information flow analysis. In: PLAS (2014)
8. Birgisson, A., Hedin, D., Sabelfeld, A.: Boosting the permissiveness of dynamic information-flow tracking by testing. In: ESORICS (2012)
9. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Programming Languages and Systems (2013)
10. Buiras, P., Stefan, D., Russo, A.: On dynamic flow-sensitive floating-label systems. In: CSF (2014)
11. Capizzi, R., Longo, A., Venkatakrishnan, V.N., Sistla, A.P.: Preventing information leaks through shadow executions. In: ACSAC (2008)
12. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: Proc. of CSF'10 (2010)
13. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for JavaScript. In: PLDI (2009)

14. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. CACM (1977)
15. Devriese, D., Piessens, F.: Non-interference through secure multi-execution. In: SSP (2010)
16. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: a web browser with flexible and precise information flow control. In: CCS (2012)
17. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. JIS (2009)
18. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: Proc. IEEE CSF. pp. 3–18 (Jun 2012)
19. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a JavaScript-like language. In: CSF (2015)
20. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: SAC (2014)
21. Just, S., Cleary, A., Shirley, B., Hammer, C.: Information Flow Analysis for JavaScript. In: Proc. ACM PLASTIC. pp. 9–18. ACM, USA (2011), http://doi.acm.org/10.1145/2093328.2093331
22. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based confidentiality monitoring. In: ASIAN (2006)
23. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly inlining of dynamic security monitors. Computers & Security 31(7), 827–843 (2012)
24. Matos, A.G.A., Santos, J.F., Rezk, T.: An information flow monitor for a core of DOM - introducing references and live primitives. In: TGC (2014)
25. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: CSF (2011)
26. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow (2001), software release http://www.cs.cornell.edu/jif
27. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proc. IEEE CSF. pp. 186–199 (Jul 2010)
28. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking information flow in dynamic tree structures. In: Proc. ESORICS. LNCS, Springer-Verlag (Sep 2009)
29. Santos, J.F., Rezk, T.: An information flow monitor-inlining compiler for securing a core of javascript. In: SEC (2014)
30. Simonet, V.: The Flow Caml system (2003), at http://cristal.inria.fr/ simonet/-soft/flowcaml
31. Stefan, D., Russo, A., Mitchell, J., Mazières, D.: Flexible dynamic information flow control in haskell. In: 4th Symposium on Haskell (2011)
32. Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: ICICS (2006)
33. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security 4(3), 167–187 (1996)