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 [15] and JSFlow [18] for JavaScript, Jif [25], Paragon [8] and JOANA [16] for Java, FlowCaml [29] for Caml, LIO [30] for Haskell, and SPARK Examiner [4] 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 [16] have been suggested to improve the precision of static information flow control, and dynamic and hybrid monitors [21, 31, 26, 18, 17] 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* [13]. 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*. Below is an example of implicit flow where the boolean value of h is leaked into l with no explicit flow involved.

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

Dynamic information flow control typically enforces *termination-insensitive non-interference* (TINI) [32]. 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 [26].

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 [1]. 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 [18], and various ways of lifting the restriction have been proposed [2, 7]. 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 [21, 31, 26, 17] 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 for example the following program:

Listing 1.1

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

Is 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.

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.

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

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

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

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.

$$\begin{array}{rll}
e ::= & x & \text{Variable} \\
\mid & l & \text{Literal} \\
\mid & x = e & \text{Assignment} \\
\mid & e \oplus e & \text{Binary Operation} \\
s ::= & e & \text{Expression Statement} \\
\mid & \texttt{skip} & \text{Null Statement} \\
\mid & \texttt{if}(e)\{s\}\{s\} & \text{Conditional} \\
\mid & \texttt{while}(e)\{s\} & \text{Loop} \\
\mid & s\,;s & \text{Sequence}
\end{array}$$

The expressions consist of literal values l, binary operators abstractly represented by $\oplus$, variables and variable assignments. The statements are built up by conditional branches, while loops, sequencing and skip, with expressions lifted into the category of statements.

$$\begin{array}{rl}
\mathcal{S} : & \mathit{string} \rightarrow \mathit{LabeledValue} \\
v \in & \mathit{Value} ::= \mathit{bool} \mid \mathit{integer} \mid \mathit{string} \mid \texttt{undef} \\
\dot{v} \in & \mathit{LabeledValue} ::= v^{\sigma} \\
C \in & \mathit{Configuration} ::= \langle \mathcal{S}, \dot{v}\rangle \mid \mathcal{S} \\
pc, \sigma, \omega \in & \mathit{Label}
\end{array}$$

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 $\perp \in \mathit{Label}$ 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 $\langle \mathcal{S}, s\rangle \rightarrow_{pc} C$ 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 Appendix A 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 $\mathcal{M}_\Delta : X \to Y \cup \Delta$ is the lifted map with default value Δ, where

$$\mathcal{M}_\Delta(x) = \begin{cases} \mathcal{M}(x) & x \in dom(\mathcal{M}) \\ \Delta & \text{otherwise} \end{cases}$$

This defines the reading operation. For writing, $\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}'$ 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:

$$\text{MUPDATE}\frac{\mathcal{M}_\Delta(x) = w^\omega \quad pc \sqsubseteq \omega}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

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. 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.

$$\text{MUPDATE-VS}\frac{\mathcal{M}_\Delta(x) = w^\omega \quad pc \not\sqsubseteq \omega \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}}$$

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 [8, 9] the labels on the values are observable. Since they change less often than the value themselves, they can be modeled as OAVs.

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, which is itself extended to handle property deletion $\mathcal{L}_{rd}$ in Section 3.3. The two latter extensions illustrate that the approach subsumes and improves previous handling of records [19].

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.

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

Listing 1.2

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.

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

Listing 1.3

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.

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:

$$\begin{array}{lll} e ::= \cdots & & \mathcal{L} \\ \quad \mid \texttt{typeof}(e) & \text{Type of} & \\ s ::= \cdots & & \mathcal{L} \end{array}$$

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 $\mathcal{T}$.

$$\begin{array}{rl} \mathcal{V} : & \mathit{string} \rightarrow \mathit{LabeledValue} \\ \mathcal{T} : & \mathit{string} \rightarrow \mathit{LabeledType} \\ \underline{v} \in & \mathit{TypedValue} ::= \langle \dot{v}, \dot{t} \rangle \\ t \in & \mathit{Type} ::= \mathit{bool} \mid \mathit{int} \mid \mathit{str} \mid \mathit{undef} \\ \mathcal{S} \in & \mathit{State} ::= \langle \mathcal{V}, \mathcal{T} \rangle \end{array}$$

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 Appendix A.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 $\bot$.

Further, the rules for variable assignment (ASSIGN_t) and variable look-up (VAR_t) require special attention. Notice that, for both maps $\mathcal{V}$ and $\mathcal{T}$, the default lookup value is undefined: $\texttt{undef}^{\bot}$ and $\mathit{undef}^{\bot}$ respectively. These maps are

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

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

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

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

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 independent existence label [27, 19, 23]. 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

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

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

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

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

that makes it possible to check if a property is present in a record.

$$\begin{array}{rrr} e ::= \cdots & & \mathcal{L} \\ & \mid \{\overline{e : e}\} & \text{Record} \\ & \mid x\,\texttt{[}e\texttt{]} & \text{Projection} \\ & \mid x\,\texttt{[}e\texttt{]} = e & \text{Property Assignment} \\ & \mid e\,\texttt{in}\,x & \text{Existence In Record} \\ s ::= \cdots & & \mathcal{L} \end{array}$$

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

$$\begin{array}{ll} \mathcal{S} : \mathit{string} \rightarrow \mathit{LabeledValue} \\ \mathcal{V} : \mathit{string} \rightarrow \mathit{LabeledValue} \\ \mathcal{E} : \mathit{string} \rightarrow \mathit{LabeledBool} \\ v \in \quad \mathit{Value} ::= r \quad \mid \quad (\cdots \text{as in } \mathcal{L}) \\ r \in \quad \mathit{Record} ::= \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma} \end{array}$$

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

```
1 o={ e_L: 0^L ,f_L: 1^M , g_H: 2^H }_H ;
2 if ( m^M ) {
3   o['e'] = 0;
4   o['h'] = 0;
5   o['f'] = 0;
6   o['g'] = 0;
7 }
```

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 3) 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 4) 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 $\{ \ldots, \mathtt{h}_M : 0^M \}_H$.

To contrast, consider the next property update (`o['f']=0`, on line 5), 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 $\{ \ldots, \mathtt{f}_L : 0^M, \ldots \}_H$.

Finally (`o['g']=0`, on line 6), 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 $\{ \ldots, \mathtt{g}_M : 0^M, \ldots \}_H$

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

$$\textsc{RecDel}\ \frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle^{\sigma_x}_{\varsigma} \quad \sigma = pc \sqcup \sigma_f \quad \mathcal{V}_1[f \xleftarrow{\texttt{undef}^{\varsigma}} \texttt{undef}^{\perp}]\downarrow_{\sigma} \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\texttt{false}^{\varsigma}} \texttt{false}^{\perp}]\downarrow_{\sigma} \mathcal{E}_2}{\langle \mathcal{S}_1, \texttt{delete}\ x\texttt{[}e\texttt{]}\rangle \rightarrow_{pc} \mathcal{S}_2[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle^{\sigma_x}_{\varsigma}]}$$

Fig. 4: $\mathcal{L}_{rd}$ semantics extension over $\mathcal{L}_r$

previous work and increases the precision of the tracking. In particular, as is the topic of the next section, this is true when property deletion is considered.

3.3 Property deletion $\mathcal{L}_{rd}$

Listing 1.6

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

Consider the initially empty record, o, in Listing 1.6. In order to be extended under secret control (line 2), the structure needs to be secret. In accordance with the previous section, after execution o will contain one property x with secret existence and value. Now, in the traditional semantics [19, 23, 27], line 3 will return the record to its initial state: empty with secret structure. This causes the (non-)existence of x, requested on line 4, to be secret, since the security label of the existence of absent properties is inherited from the structure label. Notice that the deletion on line 3 is performed under public control. Semantically, this means that the absence of the property is public and that it would be safe to label l as such. Consider an extension of $\mathcal{L}_r$ with property deletion defining $\mathcal{L}_{rd}$:

$$\begin{array}{lll} e ::= \cdots & & \mathcal{L}_r \\ s ::= \cdots & & \mathcal{L}_r \\ \quad \mid \texttt{delete}\ x\texttt{[}e\texttt{]} & \text{Delete Property} & \\ \quad \mid \texttt{delete}\ x & \text{Delete Variable} & \end{array}$$

The extension is straight forward in that it does not require any fundamental changes in the semantic modeling. The semantics for property deletion (RecDel) is found in Figure 4. The actual deletion is performed by updating the existence to register that the property no longer exists in addition to replacing the value with the default value. Hence, the label of the existence and the label of the value both reflect the security level associated with the absence of the property, analogous to the presence.

In the same way update of records is flow sensitive, so is deletion — they are both phrased in terms of the same basic primitive. This way, the security label of the absence of properties can be lowered. In particular deleting a property under public control makes its absence public.

It is interesting to note that the structure label can be seen as an abstract property of the existence map $\mathcal{E}$, and hence be modeled as an OAV, as it will be explained in the next section. Moreover, the label of the default value in the lifted

maps $\mathcal{V}$ and $\mathcal{E}$ allows value sensitivity to naturally model the structure label. Compare with [19], where the existence meta information refers to the presence while the absence was tracked by the structure label. This new representation of the structure label as the label of the default value in the affected maps uniformly models existence and absence of elements in data structure, resulting in clearer rules and proofs.

Returning to Listing 1.6, observe that the final state of the record o under this semantics is $\langle\{(\mathtt{x}, \mathtt{undef}^L)\}, \{(\mathtt{x}, \mathtt{false}^L)\}\rangle_H$. In turn, l will be $\mathtt{false}^L$, reflecting that the absence of x does not encode any secret information. This illustrates how a systematic application of the approach results in a system that outperforms previous work.

3.4 Implementation considerations: the partial order of 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. For instance, the previous section briefly discussed one such situation and argued that the structure naturally can be seen as an abstraction of the existence map. This gives rise to a partial order of OAVs based on their relative level of abstraction. As an example, consider the partial order in Figure 5 consisting of the OAVs introduced so far.

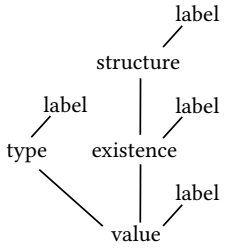

Fig. 5: OAV Lattice

Such partial order is of interest both from an implementation and proof perspective. With respect to the former, the partial order opens up for an implementation optimization based on the value-sensitivity criterion. In particular, if a less abstract OAV (in terms of the partial order) does not change there is no reason to continue the process up the chains rooted in the OAV, e.g., if the value does not change, then its more abstract notion of type will not change either. This has the potential to significantly reduce the cost of computing with longer chains of OAVs, since, in most cases, there is no need to continue past the first OAV.

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 [12, 20, 24], or code inlining [11, 5, 22, 28]. In other cases, the static analysis is triggered at runtime by the monitor [21, 31, 26, 17]. 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

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

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

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

Fig. 6: Partial hybrid semantics

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 [21] and [17], a static analysis is performed when a branching is reached.

Consider the rule for conditionals (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.

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

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

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, i.e., the value-sensitive discipline accepts more safe programs.

For space reasons, the soundness proof can be found in the Appendix B.

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 [1], $\mathcal{L}_{rd}$ is compared to the record subset of JSFlow [18] and $\mathcal{L}_h$ is compared to the Le Guernic et al.'s hybrid monitor [21].

5.1 Comparison with Austin & Flanagan's NSU [1]

The comparison with non-sensitive upgrade is relatively straight forward, since $\mathcal{L}$ is essentially the NSU monitor of [1] 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 Appendix C.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 [18]

Hedin et al. [18] 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.

Listing 1.7

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

Consider the example in Listing 1.7 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 $\langle\{\mathtt{x} \rightarrow \mathtt{0}^H\}, \{\mathtt{x} \rightarrow \mathtt{true}^H\}\rangle_H$, while for JSFlow the resulting object would be $\{\mathtt{x} \xrightarrow{H} \mathtt{0}^H\}_H$.

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

$\langle\{\mathtt{x} \rightarrow \mathtt{undef}^L\}, \{\mathtt{x} \rightarrow \mathtt{false}^L\}\rangle_H$. 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 [21]

The hybrid monitor presented by Le Guernic et al. [21] 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 Appendix C). 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 6) corresponds to monitor by Le Guernic et al.

6 Soundness

Value sensitivity is sound with respect to the common information flow policy: *termination insensitive non-interference* (TINI).

Intuitively, with respect to 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. For a general classification, the idea is the same, but rather than categorizing values into public and secret, the values are divided with respect to the level of the attacker. This way, values at or below the level of the attacker correspond to public values, and the remaining values correspond to secret values.

Let λ denote the level of the attacker. More formally, the notion of TINI can be phrased as the preservation of a λ-equivalence relation by *terminating runs* of a program. λ-equivalence formalizes the intuition that values at or below the level of the attacker are independent of values above the attacker as well as independent of unrelated values. By demanding that λ-equivalence is preserved under

execution for every label $\lambda \in \mathit{Label}$, non-interference is guaranteed regardless of the level of the attacker.

Definition 1 (termination insensitive non-interference). *Two statements s_1 and s_2 are non-interfering, $ni(s_1, s_2)$ if:*

$$ni(s_1, s_2) \stackrel{\text{def}}{=\!=} \forall \lambda \,.\, \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, s_1 \rangle \rightarrow_{pc} \mathcal{S}_2 \wedge \langle \mathcal{S}_1', s_2 \rangle \rightarrow_{pc} \mathcal{S}_2' \quad \Rightarrow \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$$

We consider a program s secure when $ni(s, s)$, meaning that it satisfies TINI. The language $\mathcal{L}$ and its derivatives presented in Section 3 are all sound with respect to TINI.

Theorem 2 ($\mathcal{L}$ and its derivatives satisfy TINI). *Let $\mathcal{L}_x$ range over the languages introduced, i.e., $\mathcal{L}_x \in \{\mathcal{L}, \mathcal{L}_t, \mathcal{L}_r, \mathcal{L}_{rd}, \mathcal{L}_h\}$. It holds that*

$$\forall s \in \mathcal{L}_x, ni(s, s)$$

Proof. By induction on the length of the execution. For space reasons the proof can be found in Appendix B.

7 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 [26] 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* [2]. In this approach, the variables assigned under high context are tagged as *partially-leaked* and cannot be used for future branching. Bichhawat et al. [6] 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 [11, 5, 22, 28], selective tracking [12, 24], and the application of a static analysis at branch points [21, 31, 26, 17]. 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. [9] extend LIO [30] 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. [23] 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. [20] only alters the structure of objects and arrays when properties or elements are inserted or removed. Similarly, Hedin et al. [17, 18] 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 [10, 14] 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 [3] 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.

8 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 [18] 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. Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *Proc. ACM PLAS* (2009).
2. Austin, T. H., and Flanagan, C. Permissive dynamic information flow analysis. In *PLAS* (2010).
3. Austin, T. H., and Flanagan, C. Multiple facets for dynamic information flow. In *PoPL* (2012).
4. Barnes, J., and Barnes, J. *High Integrity Software: The SPARK Approach to Safety and Security.* 2003.
5. Bello, L., and Bonelli, E. On-the-fly inlining of dynamic dependency monitors for secure information flow. In *FAST* (2011).
6. Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Generalizing permissive-upgrade in dynamic information flow analysis. In *PLAS* (2014).
7. Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS* (2012).
8. Broberg, N., van Delft, B., and Sands, D. Paragon for practical programming with information-flow control. In *Programming Languages and Systems*. 2013.
9. Buiras, P., Stefan, D., and Russo, A. On dynamic flow-sensitive floating-label systems. In *CSF* (2014).
10. Capizzi, R., Longo, A., Venkatakrishnan, V. N., and Sistla, A. P. Preventing information leaks through shadow executions. In *ACSAC* (2008).
11. Chudnov, A., and Naumann, D. A. Information flow monitor inlining. In *Proc. of CSF'10* (2010).
12. Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavaScript. In *PLDI* (2009).
13. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *CACM* (1977).
14. Devriese, D., and Piessens, F. Non-interference through secure multi-execution. In *SSP* (2010).
15. Groef, W. D., Devriese, D., Nikiforakis, N., and Piessens, F. Flowfox: a web browser with flexible and precise information flow control. In *CCS* (2012).
16. Hammer, C., and Snelting, G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *JIS 2009*.
17. Hedin, D., Bello, L., and Sabelfeld, A. Value-sensitive hybrid information flow control for a JavaScript-like language. In *CSF* (2015).
18. Hedin, D., Birgisson, A., Bello, L., and Sabelfeld, A. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC* (2014).
19. Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF* (June 2012), pp. 3–18.
20. Just, S., Cleary, A., Shirley, B., and Hammer, C. Information Flow Analysis for JavaScript. In *Proc. ACM PLASTIC* (USA, 2011), ACM, pp. 9–18.
21. Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *ASIAN* (2006).
22. Magazinius, J., Russo, A., and Sabelfeld, A. On-the-fly inlining of dynamic security monitors. *Computers & Security 31*, 7 (2012), 827–843.

23. Matos, A. G. A., Santos, J. F., and Rezk, T. An information flow monitor for a core of DOM - introducing references and live primitives. In *TGC* (2014).
24. Moore, S., and Chong, S. Static analysis for efficient hybrid information-flow control. In *CSF* (2011).
25. Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java information flow. Software release http://www.cs.cornell.edu/jif, 2001.
26. Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE CSF* (July 2010), pp. 186–199.
27. Russo, A., Sabelfeld, A., and Chudnov, A. Tracking information flow in dynamic tree structures. In *Proc. ESORICS* (Sept. 2009), LNCS, Springer-Verlag.
28. Santos, J. F., and Rezk, T. An information flow monitor-inlining compiler for securing a core of javascript. In *SEC* (2014).
29. Simonet, V. The Flow Caml system. At http://cristal.inria.fr/ simonet/soft/flowcaml, 2003.
30. Stefan, D., Russo, A., Mitchell, J., and Mazières, D. Flexible dynamic information flow control in haskell. In *4th Symposium on Haskell* (2011).
31. Venkatakrishnan, V. N., Xu, W., DuVarney, D. C., and Sekar, R. Provably correct runtime enforcement of non-interference properties. In *ICICS* (2006).
32. Volpano, D., Smith, G., and Irvine, C. A sound type system for secure flow analysis. *J. Computer Security 4*, 3 (1996), 167–187.

A Full set of rules

For $\mathcal{L}$, presented partially in Figure 1

$$\textsc{Var}\frac{\mathcal{S}_{\texttt{undef}^\perp}(x) = \dot{v}}{\langle \mathcal{S}, x\rangle \to_{pc} \langle \mathcal{S}, \dot{v}\rangle} \qquad \textsc{Lit}\frac{}{\langle \mathcal{S}, l\rangle \to_{pc} \langle \mathcal{S}, l^\perp\rangle}$$

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

$$\textsc{BiOp}\frac{\langle \mathcal{S}_1, e_1\rangle \to_{pc} \langle \mathcal{S}_2, v_1^{\sigma_1}\rangle \quad \langle \mathcal{S}_2, e_2\rangle \to_{pc} \langle \mathcal{S}_3, v_2^{\sigma_2}\rangle}{\langle \mathcal{S}_1, e_1 \oplus e_2\rangle \to_{pc} \langle \mathcal{S}_3, (v_1 \oplus v_2)^{\sigma_1 \sqcup \sigma_2}\rangle}$$

$$\textsc{Expr}\frac{\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, \dot{v}\rangle}{\langle \mathcal{S}_1, e\rangle \to_{pc} \mathcal{S}_2} \qquad \textsc{Skip}\frac{}{\langle \mathcal{S}, \texttt{skip}\rangle \to_{pc} \mathcal{S}}$$

$$\textsc{If}\frac{\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, v^\sigma\rangle \quad \langle \mathcal{S}_2, s_v\rangle \to_{pc \sqcup \sigma} \mathcal{S}_3}{\langle \mathcal{S}_1, \texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}\rangle \to_{pc} \mathcal{S}_3}$$

$$\textsc{While}\frac{\langle \mathcal{S}_1, \texttt{if(}e\texttt{)\{}s\texttt{;while(}e\texttt{)\{}s\texttt{\}\}\{skip\}}\rangle \to_{pc} \mathcal{S}_2}{\langle \mathcal{S}_1, \texttt{while(}e\texttt{)\{}s\texttt{\}}\rangle \to_{pc} \mathcal{S}_2}$$

$$\textsc{Seq}\frac{\langle \mathcal{S}_1, s_1\rangle \to_{pc} \mathcal{S}_2 \quad \langle \mathcal{S}_2, s_2\rangle \to_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, s_1\,\texttt{;}\,s_2\rangle \to_{pc} \mathcal{S}_3}$$

A.1 $\mathcal{L}_t$ rules

For $\mathcal{L}_t$, presented partially in Figure 2. The functions $type(\cdot)$ and $string(\cdot)$ return the type of the literal and the cast to string, respectively. The returned

value is labeled as the parameter in both cases. The statements $s;s$, `skip`, and `while(`e`){`s`}` are reduced exactly as in $\mathcal{L}$, meaning that the rules SEQ, SKIP, and WHILE respectively are applicable here.

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

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

$$\textsc{BiOp}_t \frac{\langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v_1^{\sigma_1}, t^{\sigma_3}\rangle\rangle \quad \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \langle v_2^{\sigma_2}, t^{\sigma_4}\rangle\rangle}{\langle \mathcal{S}_1, e_1 \oplus e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \langle (v_1 \oplus v_2)^{\sigma_1 \sqcup \sigma_2}, t^{\sigma_3 \sqcup \sigma_4}\rangle\rangle}$$

$$\textsc{Expr}_t \frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t}\rangle\rangle}{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \mathcal{S}_2}$$

$$\textsc{If}_t \frac{\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v^\sigma, \dot{t}\rangle\rangle \quad t = bool \quad \langle \mathcal{S}_2, s_v\rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3}{\langle \mathcal{S}_1, \texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}\rangle \rightarrow_{pc} \mathcal{S}_3}$$

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

A.2 $\mathcal{L}_r$ rules

The language $\mathcal{L}_r$ is an extension of $\mathcal{L}$. All the rules of the $\mathcal{L}$ semantics are applicable here, plus the following ones:

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

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

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

$$\textsc{Rec} \frac{\begin{array}{c}\langle \mathcal{S}_1, r\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \mathcal{V}_1, \mathcal{E}_1\rangle_\varsigma^\perp\rangle \quad \langle \mathcal{S}_2, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_3, f^{\sigma_f}\rangle \\ \langle \mathcal{S}_3, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_4, \dot{v}\rangle \quad \sigma = pc \sqcup \sigma_f \\ \mathcal{V}_1[f \xleftarrow{\texttt{undef}^\varsigma} \dot{v}]\downarrow_\sigma \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\texttt{false}^\varsigma} \texttt{true}^\perp]\downarrow_\sigma \mathcal{E}_2\end{array}}{\langle \mathcal{S}_1, r \rhd \{e_1 : e_2\}\rangle \rightarrow_{pc} \langle \mathcal{S}_4, \langle \mathcal{V}_2, \mathcal{E}_2\rangle_\varsigma^\perp\rangle}$$

$$\textsc{RecEmpty} \frac{}{\langle \mathcal{S}, \{\}\rangle \rightarrow_{pc} \langle \mathcal{S}, \langle \emptyset, \emptyset\rangle_\varsigma^\perp\rangle}$$

A.3 $\mathcal{L}_{rd}$ rules

The language $\mathcal{L}_{rd}$ is an extension over $\mathcal{L}_r$. All the rules of the $\mathcal{L}_r$ semantics are applicable here, plus the following rules:

$$\textsc{RecDel}\ \frac{\langle \mathcal{S}_1, e\rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \mathcal{S}_2(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle^{\sigma_x}_{\varsigma} \quad \sigma = pc \sqcup \sigma_f \quad \mathcal{V}_1[f \xleftarrow{\texttt{undef}^{\varsigma}} \texttt{undef}^{\perp}]\downarrow_{\sigma} \mathcal{V}_2 \quad \mathcal{E}_1[f \xleftarrow{\texttt{false}^{\varsigma}} \texttt{false}^{\perp}]\downarrow_{\sigma} \mathcal{E}_2}{\langle \mathcal{S}_1, \texttt{delete}\ x\texttt{[}e\texttt{]}\rangle \to_{pc} \mathcal{S}_2[x \to \langle \mathcal{V}_2, \mathcal{E}_2\rangle^{\sigma_x}_{\varsigma}]}$$

$$\textsc{VarDel}\ \frac{\mathcal{S}_1[x \xleftarrow{\texttt{undef}^{\perp}} \texttt{undef}^{\perp}]\downarrow_{pc} \mathcal{S}_2}{\langle \mathcal{S}_1, \texttt{delete}\ x\rangle \to_{pc} \mathcal{S}_2}$$

A.4 $\mathcal{L}_h$ rules

The syntax of $\mathcal{L}_h$ is the same as $\mathcal{L}$. Nevertheless, the semantics for branching differs (the rule If is replaced by $\textsc{If}_h$, from Figure 6), triggering the following static analysis.

$$\textsc{S-Var}\ \frac{\mathcal{S}_{\texttt{undef}^{\perp}}(x) = \dot{v}}{\langle \mathcal{S}, x\rangle \Rightarrow_{pc} \langle \mathcal{S}, \dot{v}\rangle} \qquad \textsc{S-Lit}\ \frac{}{\langle \mathcal{S}, l\rangle \Rightarrow_{pc} \langle \mathcal{S}, l^{\perp}\rangle}$$

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

$$\textsc{S-BiOp}\ \frac{\langle \mathcal{S}_1, e_1\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, v_1^{\sigma_1}\rangle \quad \langle \mathcal{S}_2, e_2\rangle \Rightarrow_{pc} \langle \mathcal{S}_3, v_2^{\sigma_2}\rangle}{\langle \mathcal{S}_1, e_1 \oplus e_2\rangle \Rightarrow_{pc} \langle \mathcal{S}_3, (v_1 \oplus v_2)^{\sigma_1 \sqcup \sigma_2}\rangle}$$

$$\textsc{S-Expr}\ \frac{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle}{\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc} \mathcal{S}_2} \qquad \textsc{S-Skip}\ \frac{}{\langle \mathcal{S}, \texttt{skip}\rangle \Rightarrow_{pc} \mathcal{S}}$$

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

$$\textsc{S-While}\ \frac{\langle \mathcal{S}_1, \texttt{if(}e\texttt{)\{}s\texttt{;while(}e\texttt{)\{}s\texttt{\}\}\{skip\}}\rangle \Rightarrow_{pc} \mathcal{S}_2}{\langle \mathcal{S}_1, \texttt{while(}e\texttt{)\{}s\texttt{\}}\rangle \Rightarrow_{pc} \mathcal{S}_2}$$

$$\textsc{S-Seq}\ \frac{\langle \mathcal{S}_1, s_1\rangle \Rightarrow_{pc} \mathcal{S}_2 \quad \langle \mathcal{S}_2, s_2\rangle \Rightarrow_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, s_1\texttt{;}\, s_2\rangle \Rightarrow_{pc} \mathcal{S}_3}$$

A.5 λ-equivalence relation

The λ-equivalence relation is a binary equivalence relation. Therefore, it is reflexive, symmetric, and transitive.

For $\mathcal{L}$ the set of rules defining the λ-equivalence relation are

$$\frac{\sigma \sqsubseteq \lambda}{v^{\sigma} \overset{\lambda}{\sim} v^{\sigma}} \qquad \frac{\sigma_1 \not\sqsubseteq \lambda \quad \sigma_2 \not\sqsubseteq \lambda}{v_1^{\sigma_1} \overset{\lambda}{\sim} v_2^{\sigma_2}} \qquad \frac{\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2 \quad \dot{v_1} \overset{\lambda}{\sim} \dot{v_2}}{\langle \mathcal{S}_1, \dot{v}_1\rangle \overset{\lambda}{\sim} \langle \mathcal{S}_2, \dot{v}_2\rangle} \qquad \frac{\mathcal{S}_1 \overset{\lambda}{\sim}_{\texttt{undef}^{\perp}} \mathcal{S}_2}{\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2}$$

In $\mathcal{L}_t$, the values are defined differently, as well as the program state:

$$\frac{\sigma \sqsubseteq \lambda \quad v_1 = v_2 \quad t_1 = t_2}{\langle v_1^\sigma, t_1^\sigma \rangle \overset{\lambda}{\sim} \langle v_2^\sigma, t_2^\sigma \rangle} \qquad \frac{\sigma_1 \sqcup \sigma_2 \not\sqsubseteq \lambda \quad \sigma_3 \sqcup \sigma_4 \not\sqsubseteq \lambda}{\langle v_1^{\sigma_1}, t_1^{\sigma_2} \rangle \overset{\lambda}{\sim} \langle v_2^{\sigma_3}, t_2^{\sigma_4} \rangle}$$

$$\frac{\mathcal{V}_1 \overset{\lambda}{\sim}_{\texttt{undef}^\perp} \mathcal{V}_2 \quad \mathcal{T}_1 \overset{\lambda}{\sim}_{undef^\perp} \mathcal{T}_2}{\langle \mathcal{V}_1, \mathcal{T}_1 \rangle \overset{\lambda}{\sim} \langle \mathcal{V}_2, \mathcal{T}_2 \rangle}$$

The records in the $\mathcal{L}_r$ extension of $\mathcal{L}$ are related in this way:

$$\frac{\sigma \sqsubseteq \lambda \quad \mathcal{V}_1 \overset{\lambda}{\sim}_{\texttt{undef}^\varsigma} \mathcal{V}_2 \quad \mathcal{E}_1 \overset{\lambda}{\sim}_{\texttt{false}^\varsigma} \mathcal{E}_2}{\langle \mathcal{V}_1, \mathcal{E}_1 \rangle_\varsigma^\sigma \overset{\lambda}{\sim} \langle \mathcal{V}_2, \mathcal{E}_2 \rangle_\varsigma^\sigma} \qquad \frac{\sigma_1 \not\sqsubseteq \lambda \quad \sigma_2 \not\sqsubseteq \lambda}{\langle \mathcal{V}_1, \mathcal{E}_1 \rangle_\varsigma^{\sigma_1} \overset{\lambda}{\sim} \langle \mathcal{V}_2, \mathcal{E}_2 \rangle_\varsigma^{\sigma_2}}$$

The lifted maps are related by a parametrized version of the relation. The parameter is the bottom element of that particular kind of map.

$$\frac{\forall x \,.\, M_{1\,\Delta}(x) \overset{\lambda}{\sim} M_{2\,\Delta}(x)}{\mathcal{M}_1 \overset{\lambda}{\sim}_\Delta \mathcal{M}_2}$$

B Soundness proof

We are going to prove that all the languages presented in the paper are secure with respect to termination insensitive non-interference (or just non-interference, for simplicity). Each proof is detailed in section B.1. Formally, a program (or generally, any statement) s is secure when $Sec(s)$ holds.

Definition 2 (Secure). *A statement s is secure, $Sec(s)$, if:*

$$Sec(s) \overset{\text{def}}{=\!=} ni_\lambda(s, s)$$

This means that a statement in a language is secure when non-interference holds for itself. In general, two statements are non-interfering when, starting with λ-equivalent states, they execute and finish in λ-equivalent states.

Definition 3 (Non-Interference at level λ). *Two statements s_1 and s_2 are non-interfering at level λ, $ni_\lambda(s_1, s_2)$, if:*

$$ni_\lambda(s_1, s_2) \overset{\text{def}}{=\!=} \quad \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, s_1 \rangle \rightarrow_{pc} \mathcal{S}_2 \wedge \langle \mathcal{S}_1', s_2 \rangle \rightarrow_{pc} \mathcal{S}_2' \quad \Rightarrow \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$$

The point λ in a lattice of security labels defines the level of clearance of the attacker , i.e. what she can observe. The λ-equivalence relation is structurally defined in Section A.5. Intuitively, an attacker that can observe up-to σ, cannot distinguish two λ-equivalent states.

In order to prove non-interference for each of the statements in each of the languages, we will use an important element called *confinement*. We say that a statement holds confinement at level λ when the state before its execution is λ-equivalent with the state at the end, given that they run in a context not lower than λ.

Definition 4 (Confinement at level λ). *The statement s holds confinement at level λ, $conf_\lambda(s)$, if:*

$$conf_\lambda(s) \overset{\text{def}}{=\!=} \quad pc \not\sqsubseteq \lambda \wedge \langle \mathcal{S}_1, s \rangle \rightarrow_{pc} \mathcal{S}_2 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$$

We need to prove confinement for $\mathcal{L}$, $\mathcal{L}_t$, $\mathcal{L}_r$, and $\mathcal{L}_{rd}$, which is done in section B.2. These enforcements are NSU-based. The case of $\mathcal{L}_h$ is different and, therefore, the confinement definition is different too.

In the hybrid context, confinement is that the variables dynamically updated in a particular statement have been upgraded by the the static analysis.

Definition 5 (Confinement at level λ with X variables upgraded). *The hybrid reduction of a statement s holds confinement at level λ after the variables X had been upgraded, $conf^X_\lambda(s)$, if:*

$$conf^X_\lambda(s) \stackrel{\text{def}}{=\!=} \quad pc \sqsubseteq \mathcal{S}_1[X] \wedge Y \subseteq X \wedge \langle \mathcal{S}_1, s\rangle \rightarrow^Y_{pc} \mathcal{S}_2 \quad \Rightarrow \mathcal{S}_1 \stackrel{\lambda}{\sim} \mathcal{S}_2$$

For this formal definition, the notation $pc \sqsubseteq \mathcal{S}[X]$ is a shortcut $pc \sqsubseteq \sigma_i$ in $x \in X\,.\,\mathcal{S}(x) = v^{\sigma_i}$. The new decoration in the reduction collects the updated variables. The lifted semantic is direct. See the following rules as examples:

$$\textsc{Assign}\frac{\langle \mathcal{S}_1, e\rangle \rightarrow^{X_1}_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}]\downarrow^{X_2}_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x = e\rangle \rightarrow^{X_1 \cup X_2}_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

$$\text{S-}\textsc{Assign}\frac{\langle \mathcal{S}_1, e\rangle \Rightarrow^{X_2}_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \mathcal{S}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}]\Downarrow^{X_2}_{pc} \mathcal{S}_3}{\langle \mathcal{S}_1, x = e\rangle \Rightarrow^{X_1 \cup X_2}_{pc} \langle \mathcal{S}_3, \dot{v}\rangle}$$

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

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

The non-interference definition (definition 3) fully applies to $\mathcal{L}_h$. Although, its static component needs to be proven auxiliary and the following definition is analogously presented.

Definition 6 (Non-Interference at level λ - static reduction). *The static reduction of a statement s is non-interfering at level λ, $ni^{\Rightarrow}_\lambda(s)$, if:*

$$ni^{\Rightarrow}_\lambda(s) \stackrel{\text{def}}{=\!=} \quad \mathcal{S}_1 \stackrel{\lambda}{\sim} \mathcal{S}'_1 \wedge v^\sigma \stackrel{\lambda}{\sim} v'^{\sigma'} \wedge \langle \mathcal{S}_1, s\rangle \Rightarrow_{pc} \mathcal{S}_2 \wedge \langle \mathcal{S}'_1, s\rangle \Rightarrow_{pc} \mathcal{S}'_2 \quad \Rightarrow \mathcal{S}_2 \stackrel{\lambda}{\sim} \mathcal{S}'_2$$

B.1 Main proofs of soundness

Theorem 3 ($\mathcal{L}$ is secure). $\forall s \in \mathcal{L}\,.\,Sec(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Variable** x. Based on the rule VAR we have to show

$$\begin{array}{l}\mathcal{S} \stackrel{\lambda}{\sim} \mathcal{S}' \wedge \mathcal{S}_{\texttt{undef}^\perp}(x) = \dot{v} \\ \qquad \wedge \mathcal{S}'_{\texttt{undef}^\perp}(x) = \dot{v}' \\ \qquad\quad \Rightarrow \langle \mathcal{S}, \dot{v}\rangle \stackrel{\lambda}{\sim} \langle \mathcal{S}', \dot{v}'\rangle\end{array}$$

By Lemma 1, $\dot{v} \overset{\lambda}{\nsim} \dot{v}'$. The result follows.

- **Case Literal** l. Based on the rule LIT we have to show

$$\mathcal{S} \overset{\lambda}{\nsim} \mathcal{S}' \quad \Rightarrow \langle \mathcal{S}, l^{\perp} \rangle \overset{\lambda}{\nsim} \langle \mathcal{S}', l^{\perp} \rangle$$

The result follows immediately.

- **Case Assignment** $x = e$. Based on the rule ASSIGN we have to show

$$\begin{array}{r} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\nsim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v} \rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}] \downarrow_{pc} \mathcal{S}_3 \\ \wedge \langle \mathcal{S}_1', e \rangle \rightarrow_{pc} \langle \mathcal{S}_2', \dot{v}' \rangle \wedge \mathcal{S}_2'[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}'] \downarrow_{pc} \mathcal{S}_3' \\ \Rightarrow \langle \mathcal{S}_3, \dot{v} \rangle \overset{\lambda}{\nsim} \langle \mathcal{S}_3', \dot{v}' \rangle \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\nsim} \mathcal{S}_2'$ and $\dot{v} \overset{\lambda}{\nsim} \dot{v}'$. By Lemma 2, $\mathcal{S}_3 \overset{\lambda}{\nsim} \mathcal{S}_3'$. The result follows.

- **Case Binary Operation** $e_1 \oplus e_2$. Based on the rule BIOP we have to show

$$\begin{array}{r} Sec(e_1) \wedge Sec(e_2) \wedge \mathcal{S}_1 \overset{\lambda}{\nsim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}_2, v_1^{\sigma_1} \rangle \wedge \langle \mathcal{S}_2, e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}_3, v_2^{\sigma_2} \rangle \\ \wedge \langle \mathcal{S}_1', e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}_2', {v'}_1^{\sigma_1'} \rangle \wedge \langle \mathcal{S}_2', e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}_3', {v'}_2^{\sigma_2'} \rangle \\ \Rightarrow \langle \mathcal{S}_3, (v_1 \oplus v_2)^{\sigma_1 \sqcup \sigma_2} \rangle \overset{\lambda}{\nsim} \langle \mathcal{S}_3', (v_1' \oplus v_2')^{\sigma_1' \sqcup \sigma_2'} \rangle \end{array}$$

By successive application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\nsim} \mathcal{S}_2'$ and then $\mathcal{S}_3 \overset{\lambda}{\nsim} \mathcal{S}_3'$. Also by successive application of the induction hypotheses, we get $_{(1)} v_1^{\sigma_1} \overset{\lambda}{\nsim} {v'}_1^{\sigma_1'}$ and $_{(2)} v_2^{\sigma_2} \overset{\lambda}{\nsim} {v'}_2^{\sigma_2'}$. There are two cases here:

case $\sigma_1 \sqcup \sigma_2 \sqsubseteq \lambda$**:** This means that $\sigma_1 \sqsubseteq \lambda$ and $\sigma_2 \sqsubseteq \lambda$. By (1) and (2), we can conclude that $\sigma_1 = \sigma_1'$, $\sigma_2 = \sigma_2'$, $v_1 = v_1'$, and $v_2 = v_2'$ respectively. Therefore, $v_1 \oplus v_2 = v_1' \oplus v_2'$ and the result follows.

case $\sigma_1 \sqcup \sigma_2 \not\sqsubseteq \lambda$**:** Then $\sigma_1' \sqcup \sigma_2' \sqsubseteq \lambda$ by (1) and (2). The result follows directly.

- **Case Expression Statement** e. Based on the rule EXPR we have to show

$$\begin{array}{r} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\nsim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v} \rangle \\ \wedge \langle \mathcal{S}_1', e \rangle \rightarrow_{pc} \langle \mathcal{S}_2', \dot{v}' \rangle \\ \Rightarrow \mathcal{S}_2 \overset{\lambda}{\nsim} \mathcal{S}_2' \end{array}$$

By application of the induction hypotheses, the result $\mathcal{S}_2 \overset{\lambda}{\nsim} \mathcal{S}_2'$ follows.

- **Case Null Statement** `skip`. Based on the rule SKIP we have to show

$$\mathcal{S} \overset{\lambda}{\nsim} \mathcal{S}' \quad \Rightarrow \mathcal{S} \overset{\lambda}{\nsim} \mathcal{S}'$$

Immediate, from the assumption.

- **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule IF we have to show

$$\begin{array}{r} Sec(e) \wedge Sec(s_{\texttt{true}}) \wedge Sec(s_{\texttt{false}}) \wedge \mathcal{S}_1 \overset{\lambda}{\nsim} \mathcal{S}_1' \\ \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^{\sigma} \rangle \wedge \langle \mathcal{S}_2, s_v \rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3 \\ \wedge \langle \mathcal{S}_1', e \rangle \rightarrow_{pc} \langle \mathcal{S}_2', {v'}^{\sigma'} \rangle \wedge \langle \mathcal{S}_2', s_{v'}' \rangle \rightarrow_{pc \sqcup \sigma'} \mathcal{S}_3' \\ \Rightarrow \mathcal{S}_3 \overset{\lambda}{\nsim} \mathcal{S}_3' \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ and (1) $v^{\sigma} \overset{\lambda}{\sim} v'^{\sigma'}$. There are two cases here:

case $pc \sqcup \sigma \sqsubseteq \lambda$**:** So $\sigma \sqsubseteq \lambda$ and, by (1), $\sigma = \sigma'$ and $v = v'$. Which means that $s_v = s'_{v'}$ (both execution take the same branch). Then, by a new application of the induction hypotheses, the result $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$ follows.

case $pc \sqcup \sigma \not\sqsubseteq \lambda$**:** By (1), $pc \sqcup \sigma' \not\sqsubseteq \lambda$. The result of Theorem 8 allows us to get $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$ and $\mathcal{S}'_2 \overset{\lambda}{\sim} \mathcal{S}'_3$. Then, by transitivity and symmetry, the result $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$ follows.

- **Case Loop** `while(`e`){`s`}`. Based on the rule WHILE, the while-loop is presented as a rewriting of the conditional statement. It is possible to apply the induction hypotheses because we are only interested in programs that terminate. This means the premise is shorter in executions steps than the conclusion and the result follows from there.
- **Case Sequence** $s_1\,;s_2$. Based on the rule SEQ we have to show

$$\begin{aligned} Sec(s_1) \wedge Sec(s_2) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge &\langle \mathcal{S}_1, s_1\rangle \rightarrow \mathcal{S}_2 \wedge \langle \mathcal{S}_2, s_2\rangle \rightarrow \mathcal{S}_3 \\ \wedge &\langle \mathcal{S}'_1, s_1\rangle \rightarrow \mathcal{S}'_2 \wedge \langle \mathcal{S}'_2, s_2\rangle \rightarrow \mathcal{S}'_3 \\ &\Rightarrow \mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3 \end{aligned}$$

By successive application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ and then $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$.

Theorem 4 ($\mathcal{L}_t$ is secure). $\forall s \in \mathcal{L}_t\ .\, Sec(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Variable** x. Based on the rule VAR_t we have to show

$$\begin{aligned} \langle \mathcal{V}, \mathcal{T}\rangle \overset{\lambda}{\sim} \langle \mathcal{V}', \mathcal{T}'\rangle \wedge &\mathcal{V}_{\mathtt{undef}^{\perp}}(x) = \dot{v} \wedge \mathcal{T}_{undef^{\perp}}(x) = \dot{t} \\ \wedge &\mathcal{V}'_{\mathtt{undef}^{\perp}}(x) = \dot{v}' \wedge \mathcal{T}'_{undef^{\perp}}(x) = \dot{t}' \\ &\Rightarrow \langle \langle \mathcal{V}, \mathcal{T}\rangle, \langle \dot{v}, \dot{t}\rangle\rangle \overset{\lambda}{\sim} \langle \langle \mathcal{V}', \mathcal{T}'\rangle, \langle \dot{v}', \dot{t}'\rangle\rangle \end{aligned}$$

By Lemma 1, $\dot{v} \overset{\lambda}{\sim} \dot{v}'$ and $\dot{t} \overset{\lambda}{\sim} \dot{t}'$. The result follows.

- **Case Literal** l. Based on the rule LIT_t we have to show

$$\mathcal{S} \overset{\lambda}{\sim} \mathcal{S}' \quad \Rightarrow \langle \mathcal{S}, \langle l^{\perp}, type(l)^{\perp}\rangle \overset{\lambda}{\sim} \langle \mathcal{S}', \langle l^{\perp}, type(l)^{\perp}\rangle$$

The result follows immediately.

- **Case Assignment** $x = e$. Based on the rule ASSIGN_t we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge &\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \langle \mathcal{V}_2, \mathcal{T}_2\rangle, \langle \dot{v}, \dot{t}\rangle\rangle \\ \wedge &\langle \mathcal{S}'_1, e\rangle \rightarrow_{pc} \langle \langle \mathcal{V}'_2, \mathcal{T}'_2\rangle, \langle \dot{v}', \dot{t}'\rangle\rangle \\ \wedge &\mathcal{V}_2[x \xleftarrow{\mathtt{undef}^{\perp}} \dot{v}]\downarrow_{pc} \mathcal{V}_3 \wedge \mathcal{T}_2[x \xleftarrow{undef^{\perp}} \dot{t}]\downarrow_{pc} \mathcal{T}_3 \\ \wedge &\mathcal{V}'_2[x \xleftarrow{\mathtt{undef}^{\perp}} \dot{v}']\downarrow_{pc} \mathcal{V}'_3 \wedge \mathcal{T}'_2[x \xleftarrow{undef^{\perp}} \dot{t}']\downarrow_{pc} \mathcal{T}'_3 \\ &\Rightarrow \langle \langle \mathcal{V}_3, \mathcal{T}_3\rangle, \langle \dot{v}, \dot{t}\rangle\rangle \overset{\lambda}{\sim} \langle \langle \mathcal{V}'_3, \mathcal{T}'_3\rangle, \langle \dot{v}', \dot{t}'\rangle\rangle \end{aligned}$$

By application of the induction hypotheses, $\langle \mathcal{V}_2, \mathcal{T}_2\rangle \overset{\lambda}{\sim} \langle \mathcal{V}'_2, \mathcal{T}'_2\rangle$ and $\langle \dot{v}, \dot{t}\rangle \overset{\lambda}{\sim} \langle \dot{v}', \dot{t}'\rangle$. By Lemma 2, $\mathcal{V}_3 \overset{\lambda}{\sim}_{\mathtt{undef}^{\perp}} \mathcal{V}'_3$ and $\mathcal{T}_3 \overset{\lambda}{\sim}_{undef^{\perp}} \mathcal{T}'_3$. The result follows.

- **Case Binary Operation** $e_1 \oplus e_2$. Based on the rule $\textsc{BiOp}_t$ we have to show

$$\begin{aligned} Sec(e_1) \wedge Sec(e_2) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge \langle \mathcal{S}_1, e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v_1^{\sigma_1}, t^{\sigma_3} \rangle \rangle \\ \wedge \langle \mathcal{S}'_1, e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}'_2, \langle {v'_1}^{\sigma'_1}, {t'}^{\sigma'_3} \rangle \rangle \\ \wedge \langle \mathcal{S}_2, e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}_3, \langle v_2^{\sigma_2}, t^{\sigma_4} \rangle \rangle \\ \wedge \langle \mathcal{S}'_2, e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}'_3, \langle {v'_2}^{\sigma'_2}, {t'}^{\sigma'_4} \rangle \rangle \\ \Rightarrow \langle \mathcal{S}_3, \langle (v_1 \oplus v_2)^{\sigma_1 \sqcup \sigma_2}, t^{\sigma_3 \sqcup \sigma_4} \rangle \rangle \overset{\lambda}{\sim} \langle \mathcal{S}'_3, \langle (v'_1 \oplus v'_2)^{\sigma'_1 \sqcup \sigma'_2}, {t'}^{\sigma'_3 \sqcup \sigma'_4} \rangle \rangle \end{aligned}$$

By successive application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ and then $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$. Also by successive application of the induction hypotheses, we get $_{(1)} v_1^{\sigma_1} \overset{\lambda}{\sim} {v'}_1^{\sigma'_1}$, $_{(2)} v_2^{\sigma_2} \overset{\lambda}{\sim} {v'}_2^{\sigma'_2}$ and $t^{\sigma_3 \sqcup \sigma_4} \overset{\lambda}{\sim} {t'}^{\sigma'_3 \sqcup \sigma'_4}$. There are two cases here:

case $\sigma_1 \sqcup \sigma_2 \sqsubseteq \lambda$**:** This means that $\sigma_1 \sqsubseteq \lambda$ and $\sigma_2 \sqsubseteq \lambda$. By (1) and (2), we can conclude that $\sigma_1 = \sigma'_1$, $\sigma_2 = \sigma'_2$, $v_1 = v'_1$, and $v_2 = v'_2$ respectively. Therefore, $v_1 \oplus v_2 = v'_1 \oplus v'_2$ and the result follows.

case $\sigma_1 \sqcup \sigma_2 \not\sqsubseteq \lambda$**:** Then $\sigma'_1 \sqcup \sigma'_2 \sqsubseteq \lambda$ by (1) and (2). The result follows directly.

- **Case Expression Statement** e. Based on the rule $\textsc{Expr}_t$ we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t} \rangle \rangle \\ \wedge \langle \mathcal{S}'_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}'_2, \langle \dot{v}', \dot{t}' \rangle \rangle \\ \Rightarrow \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2 \end{aligned}$$

By application of the induction hypotheses, the result $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ follows.

- **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule $\textsc{If}_t$ we have to show

$$\begin{aligned} Sec(e) \wedge Sec(s_{\texttt{true}}) \wedge Sec(s_{\texttt{false}}) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v^{\sigma}, \dot{bool} \rangle \rangle \\ \wedge \langle \mathcal{S}'_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}'_2, \langle {v'}^{\sigma'}, \dot{bool} \rangle \rangle \\ \wedge \langle \mathcal{S}_2, s_v \rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3 \\ \wedge \langle \mathcal{S}'_2, s'_{v'} \rangle \rightarrow_{pc \sqcup \sigma'} \mathcal{S}'_3 \\ \Rightarrow \mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3 \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ and (1) $v^{\sigma} \overset{\lambda}{\sim} {v'}^{\sigma'}$. There are two cases here:

case $pc \sqcup \sigma \sqsubseteq \lambda$**:** So $\sigma \sqsubseteq \lambda$ and, by (1), $\sigma = \sigma'$ and $v = v'$. Which means that $s_v = s'_{v'}$ (both execution take the same branch). Then, by a new application of the induction hypotheses, the result $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$ follows.

case $pc \sqcup \sigma \not\sqsubseteq \lambda$**:** By (1), $pc \sqcup \sigma' \not\sqsubseteq \lambda$. The result of Theorem 9 allows us to get $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$ and $\mathcal{S}'_2 \overset{\lambda}{\sim} \mathcal{S}'_3$. Then, by transitivity and symmetry, the result $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$ follows.

- **Case Typeof Expression** $\texttt{typeof}(e)$. Based on the rule $\textsc{Typeof}$ we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t} \rangle \rangle \\ \wedge \langle \mathcal{S}'_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}'_2, \langle \dot{v}', \dot{t}' \rangle \rangle \\ \Rightarrow \langle \mathcal{S}_2, \langle string(\dot{t}), str^{\perp} \rangle \rangle \overset{\lambda}{\sim} \langle \mathcal{S}'_2, \langle string(\dot{t}'), str^{\perp} \rangle \rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $\dot{t} \overset{\lambda}{\sim} \dot{t}'$. The result follows.

- **Rest of the cases**. The following cases:
  Case Null Statement `skip` (ruled by SKIP)
  Case Loop `while(`e`){`s`}` (ruled by WHILE)
  Case Sequence s_1`;`s_2 (ruled by SEQ)
  are analogous to Theorem 3, since $\mathcal{L}_t$ uses the same semantic as $\mathcal{L}$ for this part of its syntax.

Theorem 5 ($\mathcal{L}_r$ is secure). $\forall s \in \mathcal{L}_r \,.\, Sec(s)$

Proof. By induction on the length of the execution. Let $\delta = \texttt{undef}^{\varsigma}$ and $\epsilon = \texttt{false}^{\varsigma}$ (analogously, $\delta' = \texttt{undef}^{\varsigma'}$ and $\epsilon' = \texttt{false}^{\varsigma'}$) in the following proof by case:

- **Case Record** $\{\overline{e_1 : e_2}\}$. Based on the rules REC and RECEMPTY is demonstrated inductively on the structure of the record.
  Case empty record {} (ruled by RECEMPTY) holds immediately since:

$$\mathcal{S} \overset{\lambda}{\sim} \mathcal{S}' \quad \Rightarrow \langle \mathcal{S}, \langle \emptyset, \emptyset \rangle^{\perp}_{\perp} \rangle \overset{\lambda}{\sim} \langle \mathcal{S}', \langle \emptyset, \emptyset \rangle^{\perp}_{\perp} \rangle$$

  Case non-empty record $r \rhd \{e_1 : e_2\}$ (ruled by REC). We have to show:

$$\begin{aligned} Sec(e_1) \wedge Sec(e_2) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge & \langle \mathcal{S}_1, r \rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \mathcal{V}_1, \mathcal{E}_1 \rangle^{\perp}_{\perp} \rangle \\ \wedge & \langle \mathcal{S}_1', r \rangle \rightarrow_{pc} \langle \mathcal{S}_2', \langle \mathcal{V}_1', \mathcal{E}_1' \rangle^{\perp}_{\perp} \rangle \\ \wedge & \langle \mathcal{S}_2, e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}_3, f^{\sigma_f} \rangle \wedge \langle \mathcal{S}_3, e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}_4, \dot{v} \rangle \\ \wedge & \langle \mathcal{S}_2', e_1 \rangle \rightarrow_{pc} \langle \mathcal{S}_3', f^{\prime\sigma_f'} \rangle \wedge \langle \mathcal{S}_3', e_2 \rangle \rightarrow_{pc} \langle \mathcal{S}_4', \dot{v}' \rangle \\ \wedge & \sigma = pc \sqcup \sigma_f \wedge \sigma' = pc \sqcup \sigma_f' \\ \wedge & \mathcal{V}_1[f \overset{\delta}{\leftarrow} \dot{v}]\downarrow_{\sigma} \mathcal{V}_2 \wedge \mathcal{E}_1[f \overset{\epsilon}{\leftarrow} \texttt{true}^{\perp}]\downarrow_{\sigma} \mathcal{E}_2 \\ \wedge & \mathcal{V}_1'[f \overset{\delta}{\leftarrow} \dot{v}']\downarrow_{\sigma'} \mathcal{V}_2' \wedge \mathcal{E}_1'[f \overset{\epsilon}{\leftarrow} \texttt{true}^{\perp}]\downarrow_{\sigma'} \mathcal{E}_2' \\ & \Rightarrow \langle \mathcal{S}_4, \langle \mathcal{V}_2, \mathcal{E}_2 \rangle^{\perp}_{\perp} \rangle \overset{\lambda}{\sim} \langle \mathcal{S}_4', \langle \mathcal{V}_2', \mathcal{E}_2' \rangle^{\perp}_{\perp} \rangle \end{aligned}$$

By application of this subcase induction hypotheses $\langle \mathcal{V}_1, \mathcal{E}_1 \rangle^{\perp}_{\perp} \overset{\lambda}{\sim} \langle \mathcal{V}_1', \mathcal{E}_1' \rangle^{\perp}_{\perp}$. This means that $_{(1)}\mathcal{V}_1 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_1'$ and $_{(2)}\mathcal{E}_1 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_1'$. By successive application of the induction hypotheses, $\mathcal{S}_4 \overset{\lambda}{\sim} \mathcal{S}_4'$, $\dot{f} \overset{\lambda}{\sim} \dot{f}'$, and $\dot{v} \overset{\lambda}{\sim} \dot{v}'$. Then, by (1), (2), and Lemma 2 applied to both maps, $\mathcal{V}_2 \overset{\lambda}{\sim}_{\delta} V_2'$ and $\mathcal{E}_2 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_2'$. The result follows.

- **Case Projection** x`[`e`]`. Based on the rule PROJ we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge & \langle \mathcal{S}_1, e \rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f} \rangle \wedge \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E} \rangle^{\sigma_x}_{\varsigma} \\ & \wedge \langle \mathcal{S}_1', e \rangle \rightarrow_{pc} \langle \mathcal{S}_2', f^{\prime\sigma_f'} \rangle \wedge \mathcal{S}_2'(x) = \langle \mathcal{V}', \mathcal{E}' \rangle^{\sigma_x'}_{\varsigma'} \\ & \wedge \mathcal{V}_{\delta}(f) = \dot{v} \wedge \sigma = \sigma_x \sqcup \sigma_f \\ & \wedge \mathcal{V}'_{\delta'}(f') = \dot{v}' \wedge \sigma' = \sigma_x' \sqcup \sigma_f' \\ & \Rightarrow \langle \mathcal{S}_2, \dot{v}^{\sigma} \rangle \overset{\lambda}{\sim} \langle \mathcal{S}_2', \dot{v}'^{\sigma'} \rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $\dot{f} \overset{\lambda}{\sim} \dot{f}'$. By definition of λ-equivalence, $\langle \mathcal{V}, \mathcal{E} \rangle^{\sigma_x}_{\varsigma} \overset{\lambda}{\sim} \langle \mathcal{V}', \mathcal{E}' \rangle^{\sigma_x'}_{\varsigma'}$. This means that $\varsigma = \varsigma'$,$_{(1)}\mathcal{V}_1 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_1'$, and $_{(2)}\mathcal{E}_1 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_1'$. Let $\sigma = \sigma_x \sqcup \sigma_f$ and $\sigma' = \sigma_x' \sqcup \sigma_f'$ There are two cases from here:

case $\sigma \sqsubseteq \lambda$**:** So $\sigma' \sqsubseteq \lambda$. This means that $f = f'$. Then, by (1), (2), and Lemma 1, $\dot{v} \overset{\lambda}{\sim} \dot{v}'$. The result follows.
case $\sigma \not\sqsubseteq \lambda$**:** So $\sigma' \not\sqsubseteq \lambda$, and the result follows immediately.

- **Case Property Assignment** $x\,[e_1] = e_2$. Based on the rule RecAssign we have to show

$$\begin{aligned} Sec(e_1) \wedge Sec(e_2) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge &\langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \wedge \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle \\ &\wedge \langle \mathcal{S}_1', e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2', f'^{\sigma_f'}\rangle \wedge \langle \mathcal{S}_2', e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3', \dot{v}'\rangle \\ &\wedge \mathcal{S}_3(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \wedge \sigma = pc \sqcup \sigma_f \\ &\wedge \mathcal{S}_3'(x) = \langle \mathcal{V}_1', \mathcal{E}_1'\rangle_{\varsigma'}^{\sigma_x'} \wedge \sigma' = pc \sqcup \sigma_f' \\ &\wedge \mathcal{V}_1[f \overset{\delta}{\leftarrow} \dot{v}]\downarrow_{\sigma}\ \mathcal{V}_2 \wedge \mathcal{E}_1[f \overset{\epsilon}{\leftarrow} \texttt{true}^{\perp}]\downarrow_{\sigma}\ \mathcal{E}_2 \\ &\wedge \mathcal{V}_1'[f' \overset{\delta'}{\leftarrow} \dot{v}']\downarrow_{\sigma'}\ \mathcal{V}_2' \wedge \mathcal{E}_1'[f' \overset{\epsilon'}{\leftarrow} \texttt{true}^{\perp}]\downarrow_{\sigma'}\ \mathcal{E}_2' \\ \Rightarrow \langle \mathcal{S}_3[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle_{\varsigma}^{\sigma_x}], \dot{v}\rangle &\overset{\lambda}{\sim} \langle \mathcal{S}_3'[x \rightarrow \langle \mathcal{V}_2', \mathcal{E}_2'\rangle_{\varsigma'}^{\sigma_x'}], \dot{v}'\rangle \end{aligned}$$

By successive application of the induction hypotheses, $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_3'$, $\dot{v} \overset{\lambda}{\sim} \dot{v}'$, and $\dot{f} \overset{\lambda}{\sim} \dot{f}'$. By definition of λ-equivalence, $\langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \overset{\lambda}{\sim} \langle \mathcal{V}_1', \mathcal{E}_1'\rangle_{\varsigma'}^{\sigma_x'}$. This means that $\varsigma = \varsigma'$, (1)$\mathcal{V}_1 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_1'$, and (2)$\mathcal{E}_1 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_1'$. Let $\sigma = pc \sqcup \sigma_f$ and $\sigma' = pc \sqcup \sigma_f'$. There are two cases from here:

case $\sigma \sqsubseteq \lambda$**:** So $\sigma' \sqsubseteq \lambda$. This means that $f = f'$. Then, by (1), (2), and Lemma 2 applied to both maps, $\mathcal{V}_2 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_2'$ and $\mathcal{E}_2 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_2'$. The result follows.
case $\sigma \not\sqsubseteq \lambda$**:** So $\sigma' \not\sqsubseteq \lambda$, and the result follows immediately.

- **Case Existence In Record** $e\,\texttt{in}\,x$. Based on the rule In we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge &\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \wedge \mathcal{S}_2(x) = \langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \\ &\wedge \langle \mathcal{S}_1', e\rangle \rightarrow_{pc} \langle \mathcal{S}_2', f'^{\sigma_f'}\rangle \wedge \mathcal{S}_2'(x) = \langle \mathcal{V}', \mathcal{E}'\rangle_{\varsigma'}^{\sigma_x'} \\ &\wedge \mathcal{E}_{\epsilon}(f) = \dot{v} \wedge \sigma = \sigma_x \sqcup \sigma_f \\ &\wedge \mathcal{E}_{\epsilon'}'(f') = \dot{v}' \wedge \sigma' = \sigma_x' \sqcup \sigma_f' \\ &\Rightarrow \langle \mathcal{S}_2, \dot{v}^{\sigma}\rangle \overset{\lambda}{\sim} \langle \mathcal{S}_2', \dot{v}'^{\sigma'}\rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $\dot{f} \overset{\lambda}{\sim} \dot{f}'$. By definition of λ-equivalence, $\langle \mathcal{V}, \mathcal{E}\rangle_{\varsigma}^{\sigma_x} \overset{\lambda}{\sim} \langle \mathcal{V}', \mathcal{E}'\rangle_{\varsigma'}^{\sigma_x'}$, where $\varsigma = \varsigma'$. Let $\sigma = \sigma_x \sqcup \sigma_f$ and $\sigma' = \sigma_x' \sqcup \sigma_f'$. There are two cases from here:

case $\sigma \sqsubseteq \lambda$**:** So $\sigma' \sqsubseteq \lambda$. This means that $\mathcal{E} \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}'$ and $f = f'$. Then, by Lemma 1, $\dot{v} \overset{\lambda}{\sim} \dot{v}'$. The result follows.
case $\sigma \not\sqsubseteq \lambda$**:** So $\sigma' \not\sqsubseteq \lambda$, and the result follows immediately.

- **Rest of the cases**. The following cases:
  Case Variable x (ruled by Var)
  Case Literal l (ruled by Lit)
  Case Assignment $x = e$ (ruled by Assign)
  Case Binary Operation $e_1 \oplus e_2$ (ruled by BiOp)
  Case Expression Statement e (ruled by Expr)

Case Null Statement `skip` (ruled by SKIP)

Case Conditional $\texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}$ (ruled by IF). Theorem 10 allows the use of confinement.

Case Loop $\texttt{while(}e\texttt{)\{}s\texttt{\}}$ (ruled by WHILE)

Case Sequence $s_1\texttt{;}s_2$ (ruled by SEQ)

are analogous to Theorem 3, since $\mathcal{L}_r$ is a syntax extension of $\mathcal{L}$ with the same semantics.

Theorem 6 ($\mathcal{L}_{rd}$ is secure). $\forall s \in \mathcal{L}_{rd}\ .\ Sec(s)$

Proof. By induction on the length of the execution. Let $\delta = \texttt{undef}^{\varsigma}$ and $\epsilon = \texttt{false}^{\varsigma}$ analogously, $\delta' = \texttt{undef}^{\varsigma'}$ and $\epsilon' = \texttt{false}^{\varsigma'}$) in the following proof by case:

• **Case Delete Property** $\texttt{delete}\ x\texttt{[}e\texttt{]}$. Based on the rule RECDEL we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge &\langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \\ \wedge &\langle \mathcal{S}_1', e\rangle \rightarrow_{pc} \langle \mathcal{S}_2', f'^{\sigma'_f}\rangle \\ \wedge &\mathcal{S}_2(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \wedge \sigma = pc \sqcup \sigma_x \sqcup \sigma_f \\ \wedge &\mathcal{S}_2'(x) = \langle \mathcal{V}_1', \mathcal{E}_1'\rangle_{\varsigma'}^{\sigma'_x} \wedge \sigma' = pc \sqcup \sigma'_x \sqcup \sigma'_f \\ \wedge &\mathcal{V}_1[f \overset{\delta}{\leftarrow} \texttt{undef}^{\perp}]\downarrow_{\sigma}\ \mathcal{V}_2 \wedge \mathcal{E}_1[f \overset{\epsilon}{\leftarrow} \texttt{false}^{\perp}]\downarrow_{\sigma}\ \mathcal{E}_2 \\ \wedge &\mathcal{V}_1'[f' \overset{\delta'}{\leftarrow} \texttt{undef}^{\perp}]\downarrow_{\sigma'}\ \mathcal{V}_2' \wedge \mathcal{E}_1'[f' \overset{\epsilon'}{\leftarrow} \texttt{false}^{\perp}]\downarrow_{\sigma'}\ \mathcal{E}_2' \\ &\Rightarrow \langle \mathcal{S}_2[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle_{\varsigma}^{\sigma}], \dot{v}^{\sigma}\rangle \overset{\lambda}{\sim} \langle \mathcal{S}_2'[x \rightarrow \langle \mathcal{V}_2', \mathcal{E}_2'\rangle_{\varsigma'}^{\sigma'}], \dot{v}'^{\sigma'}\rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $\dot{f} \overset{\lambda}{\sim} \dot{f}'$. By definition of λ-equivalence, $\langle \mathcal{V}_1, \mathcal{E}_1\rangle_{\varsigma}^{\sigma_x} \overset{\lambda}{\sim} \langle \mathcal{V}_1', \mathcal{E}_1'\rangle_{\varsigma'}^{\sigma'_x}$, where $\varsigma = \varsigma'$. Let $\sigma = pc \sqcup \sigma_x \sqcup \sigma_f$ and $\sigma' = pc \sqcup \sigma'_x \sqcup \sigma'_f$. There are two cases from here:

case $\sigma \sqsubseteq \lambda$**:** So $\sigma' \sqsubseteq \lambda$. This means that $\mathcal{V}_1 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_1'$, $\mathcal{E}_1 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_1'$, and $f = f'$. Then, by Lemma 2 applied to both maps, $\mathcal{V}_2 \overset{\lambda}{\sim}_{\delta} \mathcal{V}_2'$ and $\mathcal{E}_2 \overset{\lambda}{\sim}_{\epsilon} \mathcal{E}_2'$. The result follows.

case $\sigma \not\sqsubseteq \lambda$**:** So $\sigma' \not\sqsubseteq \lambda$, and the result follows immediately.

• **Case Delete Variable** `delete` x. Based on the rule VARDEL we have to show

$$\begin{aligned} \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge &\mathcal{S}_1[x \overset{\delta}{\leftarrow} \texttt{undef}^{\perp}]\downarrow_{pc}\ \mathcal{S}_2 \\ \wedge &\mathcal{S}_1'[x \overset{\delta}{\leftarrow} \texttt{undef}^{\perp}]\downarrow_{pc}\ \mathcal{S}_2' \\ &\Rightarrow \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2' \end{aligned}$$

The result follows by Lemma 2.

• **Rest of the cases**. The following cases:

Case Variable x (ruled by VAR)

Case Literal l (ruled by LIT)

Case Assignment $x = e$ (ruled by ASSIGN)

Case Binary Operation $e_1 \oplus e_2$ (ruled by BIOP)

Case Expression Statement e (ruled by EXPR)

Case Null Statement `skip` (ruled by SKIP)

Case Conditional $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$ (ruled by IF). Theorem 11 allows the use of confinement.

Case Loop $\texttt{while}(e)\{s\}$ (ruled by WHILE)

Case Sequence $s_1\,;s_2$ (ruled by SEQ)

Case Record $\{\overline{e_1 : e_2}\}$ (ruled by REC)

Case Projection $x[e]$ (ruled by PROJ)

Case Property Assignment $x[e_1] = e_2$ (ruled by RECASSIGN)

Case Existence In Record $e\,\texttt{in}\,x$ (ruled by IN)

are analogous to Theorem 5, since $\mathcal{L}_{rd}$ is a syntax extension of $\mathcal{L}_r$ with the same semantics.

Theorem 7 ($\mathcal{L}_h$ is secure). $\forall s \in \mathcal{L}_h \,.\, Sec(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Assignment** $x = e$. Based on the rule ASSIGN we have to show

$$\begin{aligned} Sec(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \wedge \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}]{\downarrow_{pc}}\, \mathcal{S}_3 \\ \wedge \langle \mathcal{S}_1', e\rangle \rightarrow_{pc} \langle \mathcal{S}_2', \dot{v}'\rangle \wedge \mathcal{S}_2'[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}']{\downarrow_{pc}}\, \mathcal{S}_3' \\ \Rightarrow \langle \mathcal{S}_3, \dot{v}\rangle \overset{\lambda}{\sim} \langle \mathcal{S}_3', \dot{v}'\rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $\dot{v} \overset{\lambda}{\sim} \dot{v}'$. By Lemma 9, $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_3'$. The result follows.

- **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule IF_h we have to show

$$\begin{aligned} Sec(e) \wedge Sec(s_{\texttt{true}}) \wedge\, & Sec(s_{\texttt{false}}) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_1' \\ & \wedge \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^{\sigma}\rangle \wedge \langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_t \\ & \wedge \langle \mathcal{S}_1', e\rangle \rightarrow_{pc} \langle \mathcal{S}_2', v'^{\sigma'}\rangle \wedge \langle \mathcal{S}_2', s_{\texttt{true}}\rangle \Rightarrow_{pc\sqcup\sigma'} \mathcal{S}_t' \\ & \wedge \langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow_{pc\sqcup\sigma} \mathcal{S}_f \wedge \langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_v\rangle \rightarrow_{pc\sqcup\sigma} \mathcal{S}_3 \\ & \wedge \langle \mathcal{S}_2', s_{\texttt{false}}\rangle \Rightarrow_{pc\sqcup\sigma'} \mathcal{S}_f' \wedge \langle \mathcal{S}_t' \sqcup \mathcal{S}_f', s_{v'}\rangle \rightarrow_{pc\sqcup\sigma'} \mathcal{S}_3' \\ & \Rightarrow \mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_3' \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_2'$ and $_{(1)}v^{\sigma} \overset{\lambda}{\sim} v'^{\sigma'}$. The result of the Lemma 4 give us $\mathcal{S}_t \overset{\lambda}{\sim} \mathcal{S}_t'$ and $\mathcal{S}_f \overset{\lambda}{\sim} \mathcal{S}_f'$. Let $\mathcal{S}_b = \mathcal{S}_t \sqcup \mathcal{S}_f$ and $\mathcal{S}_b' = \mathcal{S}_t' \sqcup \mathcal{S}_f'$. The application of Lemma 10 results in $\mathcal{S}_b \overset{\lambda}{\sim} \mathcal{S}_b'$.

The static reductions $\langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow^{X_f}_{pc\sqcup\sigma} \mathcal{S}_f$ and $\langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow^{X_t}_{pc\sqcup\sigma} \mathcal{S}_t$ gives that $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_f[X_f]$ and $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_t[X_t]$ by Lemma 7. Analogously, $\langle \mathcal{S}_2', s_{\texttt{false}}\rangle \Rightarrow^{X_f'}_{pc\sqcup\sigma} \mathcal{S}_f'$ and $\langle \mathcal{S}_2', s_{\texttt{true}}\rangle \Rightarrow^{X_t'}_{pc\sqcup\sigma} \mathcal{S}_t'$ gives that $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_f'[X_f']$ and $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_t'[X_t']$. By Lemma 11, $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_b[X_t]$ and $(pc \sqcup \sigma) \sqsubseteq \mathcal{S}_b[X_f]$. The result of Lemma 6 allows us to apply Theorem 12 to conclude that $\mathcal{S}_b \overset{\lambda}{\sim} \mathcal{S}_3$ and $\mathcal{S}_b' \overset{\lambda}{\sim} \mathcal{S}_3'$ independently of v. Then, by transitivity and symmetry, the result $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_3'$ follows.

- **Rest of the cases**. The following cases:

  Case Variable x (ruled by VAR)

  Case Literal l (ruled by LIT)

Case Binary Operation $e_1 \oplus e_2$ (ruled by BiOp)
Case Expression Statement e (ruled by Expr)
Case Null Statement `skip` (ruled by Skip)
Case Loop `while(`e`){`s`}` (ruled by While)
Case Sequence $s_1\,;s_2$ (ruled by Seq)
are analogous to Theorem 3.

B.2 Confinement

Theorem 8 ($\mathcal{L}$ holds confinement). $\forall s \in \mathcal{L}\,.\,conf_\lambda(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Variable** x. Based on the rule Var, the result follows directly by reflexivity of λ-equivalence.
- **Case Literal** l. Based on the rule Lit, the result follows directly by reflexivity of λ-equivalence.
- **Case Assignment** $x = e$. Based on the rule Assign we have to show

$$\begin{array}{c} conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}]\downarrow_{pc} \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$. By Lemma 3, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. The result follows by transitivity.

- **Case Binary Operation** $e_1 \oplus e_2$. Based on the rule BiOp we have to show

$$\begin{array}{c} conf_\lambda(e_1) \wedge conf_\lambda(e_2) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v_1^{\sigma_1}\rangle \wedge \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, v_2^{\sigma_2}\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By successive application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$ and then $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. The result follows by transitivity.

- **Case Expression Statement** e. Based on the rule Expr we have to show

$$\begin{array}{c} conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2 \end{array}$$

The result follows by the induction hypotheses.

- **Case Null Statement** `skip`. Based on the rule Skip, the result follows directly by reflexivity of λ-equivalence.
- **Case Conditional** `if(`e`){`$s_{\texttt{true}}$`}{`$s_{\texttt{false}}$`}`. Based on the rule If we have to show

$$\begin{array}{c} conf_\lambda(e) \wedge conf_\lambda(s_{\texttt{true}}) \wedge conf_\lambda(s_{\texttt{false}}) \wedge (1) pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, v^\sigma\rangle \wedge \langle \mathcal{S}_2, s_v\rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$. By (1), $pc \sqcup \sigma \not\sqsubseteq \lambda$, which means that the induction hypotheses can be applied again and $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. Then, by transitivity, the result follows.

• **Case Loop** $\texttt{while}(e)\{s\}$. The while-loop is presented in the rule WHILE as a rewriting of the conditional statement. It is possible to apply the induction hypotheses because we are only interested in programs that terminate. This means the premise is shorter in executions steps than the conclusion and the result follows from there.

• **Case Sequence** $s_1\,;s_2$. Based on the rule SEQ we have to show

$$\begin{array}{r} conf_\lambda(s_1) \wedge conf_\lambda(s_2) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, s_1\rangle \rightarrow \mathcal{S}_2 \wedge \langle \mathcal{S}_2, s_2\rangle \rightarrow \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By successive application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$ and then $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. The result follows by transitivity.

Theorem 9 ($\mathcal{L}_t$ holds confinement). $\forall s \in \mathcal{L}_t\ .conf_\lambda(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

• **Case Variable** x. Based on the rule VAR_t, the result follows directly by reflexivity of λ-equivalence.

• **Case Literal** l. Based on the rule LIT_t, the result follows directly by reflexivity of λ-equivalence.

• **Case Assignment** $x = e$. Based on the rule ASSIGN_t we have to show

$$\begin{array}{r} conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle\langle \mathcal{V}_2, \mathcal{T}_2\rangle, \langle \dot{v}, \dot{t}\rangle\rangle \wedge \\ \mathcal{V}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}]{\downarrow_{pc}}\ \mathcal{V}_3 \wedge \\ \mathcal{T}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{t}]{\downarrow_{pc}}\ \mathcal{T}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \langle \mathcal{V}_3, \mathcal{T}_3\rangle \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \langle \mathcal{V}_2, \mathcal{T}_2\rangle$. By Lemma 3, $\mathcal{V}_2 \overset{\lambda}{\sim}_{\texttt{undef}^\perp} \mathcal{V}_3$ and $\mathcal{T}_2 \overset{\lambda}{\sim}_{\texttt{undef}^\perp} \mathcal{T}_3$. The result follows by transitivity.

• **Case Binary Operation** $e_1 \oplus e_2$. Based on the rule BIOP_t we have to show

$$\begin{array}{r} conf_\lambda(e_1) \wedge conf_\lambda(e_2) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v_1^{\sigma_1}, t^{\sigma_3}\rangle\rangle \wedge \\ \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \langle v_2^{\sigma_2}, t^{\sigma_4}\rangle\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By successive application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$ and then $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. The result follows by transitivity.

• **Case Expression Statement** e. Based on the rule EXPR_t we have to show

$$\begin{array}{r} conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t}\rangle\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2 \end{array}$$

The result follows by the induction hypotheses.

• **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule IF_t we have to show

$$\begin{array}{r} conf_\lambda(e) \wedge conf_\lambda(s_{\texttt{true}}) \wedge conf_\lambda(s_{\texttt{false}}) \wedge {}_{(1)} pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle v^\sigma, \dot{t}\rangle\rangle \wedge t = bool \wedge \\ \langle \mathcal{S}_2, s_v\rangle \rightarrow_{pc \sqcup \sigma} \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{array}$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$. By (1), $pc \sqcup \sigma \not\sqsubseteq \lambda$, which means that the induction hypotheses can be applied again and $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$. Then, by transitivity, the result follows.

- **Case Typeof Expression** `typeof(`e`)`. Based on the rule TYPEOF we have to show

$$\begin{array}{r} conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \dot{v}, \dot{t}\rangle\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2 \end{array}$$

The result follows by the induction hypotheses.

- **Rest of the cases**. The following cases:
  Case Null Statement `skip` (ruled by SKIP)
  Case Loop `while(`e`){`s`}` (ruled by WHILE)
  Case Sequence s_1`;`s_2 (ruled by SEQ)
  are analogous to Theorem 8, since $\mathcal{L}_t$ uses the same semantic as $\mathcal{L}$ for this part of its syntax.

Theorem 10 ($\mathcal{L}_r$ holds confinement). $\forall s \in \mathcal{L}_r \ . conf_\lambda(s)$

Proof. By induction on the length of the execution. Let $\delta = \texttt{undef}^\varsigma$ and $\epsilon = \texttt{false}^\varsigma$ in the following proof by case:

- **Case Record** $\{\overline{e_1 : e_2}\}$.
  Based on the rules REC and RECEMPTY is demonstrated inductively on the structure of the record.
  Case empty record {} (ruled by RECEMPTY) holds immediately by reflexivity.
  Case non-empty record $r \rhd \{e_1 : e_2\}$ (ruled by REC). We have to show:

$$\begin{array}{r} conf_\lambda(e_1) \wedge conf_\lambda(e_2) \wedge pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, r\rangle \rightarrow_{pc} \langle \mathcal{S}_2, \langle \mathcal{V}_1, \mathcal{E}_1\rangle^{\perp}_{\perp}\rangle \wedge \\ \langle \mathcal{S}_2, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_3, f^{\sigma_f}\rangle \wedge \langle \mathcal{S}_3, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_4, \dot{v}\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_4 \end{array}$$

By application of this subcase induction hypotheses $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$. By successive application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3$ and $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_4$. The result follows by transitivity.

- **Case Projection** x`[`e`]`. Based on the rule PROJ we have to show

$$conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \langle \mathcal{S}_1, e\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$$

By application of the induction hypotheses, the result follows.

- **Case Property Assignment** x`[`e_1`] =`e_2. Based on the rule RECASSIGN we have to show

$$\begin{array}{r} conf_\lambda(e_1) \wedge conf_\lambda(e_2) \wedge {}_{(1)} pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e_1\rangle \rightarrow_{pc} \langle \mathcal{S}_2, f^{\sigma_f}\rangle \wedge \langle \mathcal{S}_2, e_2\rangle \rightarrow_{pc} \langle \mathcal{S}_3, \dot{v}\rangle \wedge \\ {}_{(2)}\mathcal{S}_3(x) = \langle \mathcal{V}_1, \mathcal{E}_1\rangle^{\sigma_x}_{\varsigma} \wedge \sigma = pc \sqcup \sigma_x \sqcup \sigma_f \wedge \\ \mathcal{V}_1[f \overset{\delta}{\leftarrow} \dot{v}]{\downarrow_\sigma}\ \mathcal{V}_2 \wedge \mathcal{E}_1[f \overset{\epsilon}{\leftarrow} \texttt{true}^{\perp}]{\downarrow_\sigma}\ \mathcal{E}_2 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3[x \rightarrow \langle \mathcal{V}_2, \mathcal{E}_2\rangle^{\sigma}_{\varsigma}] \end{array}$$

By successive application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3$. Since (1), $\sigma \not\sqsubseteq \lambda$ and Lemma 2 can be applied to both maps, then $\mathcal{V}_1 \overset{\lambda}{\sim}_\delta \mathcal{V}_2$ and $\mathcal{E}_1 \overset{\lambda}{\sim}_\epsilon \mathcal{E}_2$. This last result, plus (2), gives us $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}_3[x \to \langle \mathcal{V}_2, \mathcal{E}_2 \rangle^\sigma_\varsigma]$. The result follows by transitivity.

- **Case Existence In Record** $e\,\texttt{in}\,x$. Based on the rule In we have to show

$$conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \langle \mathcal{S}_1, e \rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f} \rangle \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$$

By application of the induction hypotheses, the result follows.

- **Rest of the cases**. The following cases:
  Case Variable x (ruled by Var)
  Case Literal l (ruled by Lit)
  Case Assignment $x = e$ (ruled by Assign)
  Case Binary Operation $e_1 \oplus e_2$ (ruled by BiOp)
  Case Expression Statement e (ruled by Expr)
  Case Null Statement `skip` (ruled by Skip)
  Case Conditional $\texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}$ (ruled by If)
  Case Loop $\texttt{while(}e\texttt{)\{}s\texttt{\}}$ (ruled by While)
  Case Sequence $s_1\,\texttt{;}\,s_2$ (ruled by Seq)
  are analogous to Theorem 8, since $\mathcal{L}_r$ is a syntax extension of $\mathcal{L}$ with the same semantics.

Theorem 11 ($\mathcal{L}_{rd}$ holds confinement). $\forall s \in \mathcal{L}_{rd}\,.\,conf_\lambda(s)$

Proof. By induction on the length of the execution. Let $\delta = \texttt{undef}^\varsigma$ and $\epsilon = \texttt{false}^\varsigma$ in the following proof by case:

- **Case Delete Property** $\texttt{delete}\ x\texttt{[}e\texttt{]}$. Based on the rule RecDel we have to show

$$\begin{array}{r} conf_\lambda(e) \wedge {}_{(1)}pc \not\sqsubseteq \lambda \wedge \\ \langle \mathcal{S}_1, e \rangle \to_{pc} \langle \mathcal{S}_2, f^{\sigma_f} \rangle \wedge \\ {}_{(2)}\mathcal{S}_2(x) = \langle \mathcal{V}_1, \mathcal{E}_1 \rangle^{\sigma_x}_\varsigma \wedge \sigma = pc \sqcup \sigma_x \sqcup \sigma_f \wedge \\ \mathcal{V}_1[f \overset{\delta}{\leftarrow} \texttt{undef}^\perp]\downarrow_\sigma \mathcal{V}_2 \wedge \mathcal{E}_1[f \overset{\epsilon}{\leftarrow} \texttt{false}^\perp]\downarrow_\sigma \mathcal{E}_2 \end{array} \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2[x \to \langle \mathcal{V}_2, \mathcal{E}_2 \rangle^\sigma_\varsigma]$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$. Since (1), $\sigma \not\sqsubseteq \lambda$ and Lemma 2 can be applied to both maps, then $\mathcal{V}_1 \overset{\lambda}{\sim}_\delta \mathcal{V}_2$ and $\mathcal{E}_1 \overset{\lambda}{\sim}_\epsilon \mathcal{E}_2$. This last result, plus (2), gives us $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_3[x \to \langle \mathcal{V}_2, \mathcal{E}_2 \rangle^\sigma]$. The result follows by transitivity.

- **Case Delete Variable** $\texttt{delete}\ x$. Based on the rule VarDel we have to show

$$conf_\lambda(e) \wedge pc \not\sqsubseteq \lambda \wedge \mathcal{S}_1[x \xleftarrow{\texttt{undef}^\perp} \texttt{undef}^\perp]\downarrow_{pc} \mathcal{S}_2 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$$

The result follows by Lemma 3.

- **Rest of the cases**. The following cases:
  Case Variable x (ruled by Var)
  Case Literal l (ruled by Lit)
  Case Assignment $x = e$ (ruled by Assign)
  Case Binary Operation $e_1 \oplus e_2$ (ruled by BiOp)

Case Expression Statement e (ruled by Expr)
Case Null Statement `skip` (ruled by Skip)
Case Conditional $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$ (ruled by If)
Case Loop $\texttt{while}(e)\{s\}$ (ruled by While)
Case Sequence $s_1\,;s_2$ (ruled by Seq)
Case Record $\{\overline{e_1 : e_2}\}$ (ruled by Rec)
Case Projection $x[e]$ (ruled by Proj)
Case Property Assignment $x[e_1] = e_2$ (ruled by RecAssign)
Case Existence In Record $e\,\texttt{in}\,x$ (ruled by In)
are analogous to Theorem 10, since $\mathcal{L}_{rd}$ is a syntax extension of $\mathcal{L}_r$ with the same semantics.

Theorem 12 ($\mathcal{L}_h$ holds confinement). $\forall s \in \mathcal{L}_h \,.\, conf_\lambda^X(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Assignment** $x = e$. Based on the rule Assign we have to show

$$conf_\lambda^X(e) \wedge {}_{(1)}pc \sqsubseteq \mathcal{S}_1[X] \wedge (Y_1 \cup Y_2) \subseteq X \wedge \langle \mathcal{S}_1, e\rangle \rightarrow_{pc}^{Y_1} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}]\downarrow_{pc}^{Y_2} \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3$$

By application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$ and $Y_1 \subseteq X$. Let $\mathcal{S}_{2_{\texttt{undef}^\perp}}(x) = \dot{w}$. We have two cases:

case $w = v$**:** The rule MUpdate-VS$_h$ is applied and, as a consequence, $\mathcal{S}_2 = \mathcal{S}_3$ and $Y_2 = \emptyset$. The result follows immediately.

case $w \neq v$**:** The rule MUpdate$_h$ is applied. Therefore, $Y_2 = \{x\}$ and, by (1), $pc \sqsubseteq x$. Then $\mathcal{S}_2 \overset{\lambda}{\sim} S_3$, since the update $\mathcal{S}_3 = \mathcal{S}_2[x \mapsto \dot{v}^{pc}]$ affects only the secret part of $\mathcal{S}_2$. The result follows by transitivity.

- **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule If$_h$ we have to show.

$$\begin{aligned} conf_\lambda^X(e) \wedge conf_\lambda^X(s_{\texttt{true}}) \wedge conf_\lambda^X(s_{\texttt{false}}) \wedge \\ pc \sqsubseteq \mathcal{S}_1[X] \wedge (Y_1 \cup Y_2 \cup Y_3 \cup Y_4) \subseteq X \wedge \\ \langle \mathcal{S}_1, e\rangle \rightarrow_{pc}^{Y_1} \langle \mathcal{S}_2, v^\sigma\rangle \wedge {}_{(1)}\langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow_{pc\sqcup\sigma}^{Y_2} \mathcal{S}_t \wedge \\ {}_{(2)}\langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow_{pc\sqcup\sigma}^{Y_3} \mathcal{S}_f \wedge \langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_v\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_3 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_3 \end{aligned}$$

By successive application of the induction hypotheses, $\mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_t \overset{\lambda}{\sim} \mathcal{S}_f$ and $(Y_1 \cup Y_2 \cup Y_3) \subseteq X$. The Lemma 10 give us $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}_t \sqcup \mathcal{S}_f$. We have two cases:

case $v = \texttt{true}$**:** In this case, $\langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_{\texttt{true}}\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_3$. By (1), it is possible to apply the induction hypotheses which give us $Y_2 \subseteq X$. Then, the result follows by transitivity.

case $v = \texttt{false}$**:** In this case, $\langle \mathcal{S}_t \sqcup \mathcal{S}_f, s_{\texttt{false}}\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_3$. By (2), it is possible to apply the induction hypotheses which give us $Y_3 \subseteq X$. Then, the result follows by transitivity.

- **Rest of the cases**. The following cases:
  Case Variable x (ruled by VAR)
  Case Literal l (ruled by LIT)
  Case Binary Operation $e_1 \oplus e_2$ (ruled by BIOP)
  Case Expression Statement e (ruled by EXPR)
  Case Null Statement `skip` (ruled by SKIP)
  Case Loop `while(`e`){`s`}` (ruled by WHILE)
  Case Sequence s_1`;`s_2 (ruled by SEQ)
  are direct application of the induction hypotheses.

B.3 Supporting Lemmas

Lemma 1 (Soundness of looking up in a lifted map).

$$(1)\mathcal{M} \overset{\lambda}{\sim}_{\Delta} \mathcal{M}' \wedge \mathcal{M}_{\Delta}(x) = \dot{v} \wedge \mathcal{M}'_{\Delta}(x) = \dot{v}' \quad \Rightarrow \dot{v} \overset{\lambda}{\sim} \dot{v}'$$

Proof. Based on:

$$\mathcal{M}_{\Delta}(x) = \begin{cases} \mathcal{M}(x) & x \in Dom(\mathcal{M}) \\ \Delta & \text{otherwise} \end{cases}$$

Immediate, by the definition of $M \overset{\lambda}{\sim}_{\Delta} M'$ in the assumption (1).

Lemma 2 (Soundness of MUpdate).

$$(1)\mathcal{M}_1 \overset{\lambda}{\sim}_{\Delta} \mathcal{M}'_1 \wedge (2)\dot{v} \overset{\lambda}{\sim} \dot{v}' \wedge \mathcal{M}_1[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}_2 \wedge \mathcal{M}'_1[x \xleftarrow{\Delta} \dot{v}']\downarrow_{pc} \mathcal{M}'_2 \quad \Rightarrow \mathcal{M}_2 \overset{\lambda}{\sim}_{\Delta} \mathcal{M}'_2$$

Proof. Based on the rules

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

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

Let $\mathcal{M}_{\Delta}(x) = w^{\omega}$ and $\mathcal{M}'_{\Delta}(x) = w'^{\omega'}$. By (1) and Lemma 1, (3) $w^{\omega} \overset{\lambda}{\sim} w'^{\omega'}$. We have the following two cases.

case $pc \sqsubseteq \omega$**:** So, $pc \sqsubseteq \omega'$ by (3). The rule MUPDATE is applied to both updates. By (2) we know $\dot{v}^{pc} \overset{\lambda}{\sim} \dot{v}'^{pc}$. From here, the result $\mathcal{M}[x \mapsto \dot{v}^{pc}] \overset{\lambda}{\sim}_{\Delta} \mathcal{M}'[x \mapsto \dot{v}'^{pc}]$ follows.

case $pc \not\sqsubseteq \omega$**:** So, $pc \not\sqsubseteq \omega'$ by (3). The result follows directly from the assumption (1) and the application of MUPDATE-VS.

Lemma 3 (Confinement of MUpdate).

$$(1)pc \not\sqsubseteq \lambda \wedge \mathcal{M}_1[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}_2 \quad \Rightarrow \mathcal{M}_1 \overset{\lambda}{\sim}_{\Delta} \mathcal{M}_2$$

Proof. Based on the rules

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

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

Let $\mathcal{M}_{1\,\Delta}(x) = w^{\omega}$. We have the following two cases.

case $pc \sqsubseteq \omega$**:** The rule MUpdate is applied in this case. By (1), $\omega \not\sqsubseteq \lambda$. This means that $w^{\omega} \not\overset{\lambda}{\sim} \dot{v}^{pc}$. Then $\mathcal{M}_1 \overset{\lambda}{\sim}_{\Delta} \mathcal{M}_1[x \mapsto \dot{v}^{pc}]$ and the result follows.

case $pc \not\sqsubseteq \omega$**:** The rule MUpdate-VS is applied in this case. The result follows directly by reflexivity of λ-equivalence.

Lemma 4 (The static part of $\mathcal{L}_h$ holds non-interference). $\forall s \in \mathcal{L}_h \; . ni^{\Rightarrow}_{\lambda}(s)$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Static Assignment** $x = e$. Based on the rule S-Assign we have to show

$$\begin{aligned} ni^{\Rightarrow}_{\lambda}(e) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \wedge \langle \mathcal{S}_1, e\rangle \Rightarrow^{\sigma}_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}]\Downarrow^{\sigma}_{pc} \mathcal{S}_3 \\ \wedge \langle \mathcal{S}'_1, e\rangle \Rightarrow^{\sigma'}_{pc} \langle \mathcal{S}'_2, \dot{v}'\rangle \wedge \mathcal{S}'_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}']\Downarrow^{\sigma'}_{pc} \mathcal{S}'_3 \\ \Rightarrow \langle \mathcal{S}_3, \dot{v}\rangle \overset{\lambda}{\sim} \langle \mathcal{S}'_3, \dot{v}'\rangle \end{aligned}$$

By application of the induction hypotheses, $\mathcal{S}_2 \overset{\lambda}{\sim} \mathcal{S}'_2$ and $\dot{v} \overset{\lambda}{\sim} \dot{v}'$. By Lemma 8, $\mathcal{S}_3 \overset{\lambda}{\sim} \mathcal{S}'_3$. The result follows.

- **Case Static Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule S-If we have to show

$$\begin{aligned} ni^{\Rightarrow}_{\lambda}(e) \wedge ni^{\Rightarrow}_{\lambda}(s_{\texttt{true}}) \wedge ni^{\Rightarrow}_{\lambda}(s_{\texttt{false}}) \wedge \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}'_1 \\ \wedge \langle \mathcal{S}_1, e\rangle \Rightarrow^{\sigma}_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \\ \wedge \langle \mathcal{S}'_1, e\rangle \Rightarrow^{\sigma'}_{pc} \langle \mathcal{S}'_2, \dot{v}'\rangle \\ \wedge \langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow^{\sigma}_{pc} \mathcal{S}_t \wedge \langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow^{\sigma}_{pc} \mathcal{S}_f \\ \wedge \langle \mathcal{S}'_2, s_{\texttt{true}}\rangle \Rightarrow^{\sigma'}_{pc} \mathcal{S}'_t \wedge \langle \mathcal{S}'_2, s_{\texttt{false}}\rangle \Rightarrow^{\sigma'}_{pc} \mathcal{S}'_f \\ \Rightarrow \mathcal{S}_t \sqcup \mathcal{S}_f \overset{\lambda}{\sim} \mathcal{S}'_t \sqcup \mathcal{S}'_f \end{aligned}$$

By successive application of the induction hypotheses, $\mathcal{S}_t \overset{\lambda}{\sim} \mathcal{S}'_t$ and $\mathcal{S}_f \overset{\lambda}{\sim} \mathcal{S}'_f$. The result follows by Lemma 10.

- **Rest of the cases**. The following cases:
  Case Static Variable x (ruled by S-Var, analogous to Var)
  Case Static Literal l (ruled by S-Lit, analogous to Lit)
  Case Static Null Statement `skip` (ruled by S-Skip, analogous to Skip)
  have an analogous case in the Theorem 3.
  The following cases:
  Case Static Binary Operation $e \oplus e$ (ruled by S-BiOp)
  Case Static Expression Statement e (ruled by S-Expr)
  Case Static Loop $\texttt{while}(e)\{s\}$ (ruled by S-While)

Case Static Sequence $s_1\,;s_2$ (ruled by S-SEQ)

are immediate by successive application of the induction hypothesis in each case.

Lemma 5 (The static part of $\mathcal{L}_h$ holds confinement).

$$\forall s \in \mathcal{L}_h \;.\, pc \sqsubseteq \mathcal{S}_1[X] \wedge Y \subseteq X \wedge \langle \mathcal{S}_1, s\rangle \Rightarrow^{Y}_{pc} \mathcal{S}_2 \quad \Rightarrow \mathcal{S}_1 \overset{\lambda}{\sim} \mathcal{S}_2$$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Static Assignment** $x = e$. Based on the rule S-ASSIGN, this case is analogous on the assignment case in Theorem 12.
- **Rest of the cases**. The following cases:
  Case Static Variable x (ruled by S-VAR)
  Case Static Literal l (ruled by S-LIT)
  Case Static Binary Operation $e \oplus e$ (ruled by S-BIOP)
  Case Static Expression Statement e (ruled by S-EXPR)
  Case Static Null Statement `skip` (ruled by S-SKIP)
  Case Static Conditional $\texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}$ (ruled by S-IF)
  Case Static Loop $\texttt{while(}e\texttt{)\{}s\texttt{\}}$ (ruled by S-WHILE)
  Case Static Sequence $s_1\,;s_2$ (ruled by S-SEQ)

  are immediate by successive application of the induction hypothesis in each case.

Lemma 6 (The variables modified in dynamic time, have been updated by the static analysis).

$$\langle \mathcal{S}_1, s\rangle \Rightarrow^{X}_{pc} \mathcal{S}_2 \quad \Rightarrow \forall \mathcal{S}_0 \,.\, \langle \mathcal{S}_2 \sqcup \mathcal{S}_0, s\rangle \rightarrow^{Y}_{pc} S_3 \Rightarrow Y \subseteq X$$

Proof. By induction on the length of the execution. Following, the proof by case:

- **Case Assignment** $x = e$. Based on the rule S-ASSIGN, we have $\langle \mathcal{S}_1, e\rangle \Rightarrow^{X_1}_{pc} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}] \Downarrow^{X_2}_{pc} \mathcal{S}_3$.

  The hybrid reduction is ruled by ASSIGN. We have to show:

$$\forall \mathcal{S}_0 \,.\, \langle \mathcal{S}_3 \sqcup \mathcal{S}_0, e\rangle \rightarrow^{Y_1}_{pc} \langle \mathcal{S}_4, \dot{v}\rangle \wedge \mathcal{S}_4[x \xleftarrow{\texttt{undef}^{\perp}} \dot{v}] \downarrow^{Y_2}_{pc} \mathcal{S}_5 \quad \Rightarrow (Y_1 \cup Y_2) \subseteq (X_1 \cup X_2)$$

By application of the induction hypotheses, $Y_1 \subseteq X_1$. Let $\mathcal{S}_{2_{\texttt{undef}^{\perp}}}(x) = \dot{w}$. Since $\mathcal{S}_4$ is equal to $\mathcal{S}_2$ with respect to their pure values, $\mathcal{S}_{4_{\texttt{undef}^{\perp}}}(x) = \dot{w}$. We have two cases:

case $w = v$**:** The rule S-MUPDATEVS is applied. As a consequence, $X_2 = \emptyset$ and $\mathcal{S}_2 = \mathcal{S}_3$. The rule MUPDATE-VS$_h$ give us $Y_2 = \emptyset$ and the result follows.

case $w \neq v$**:** The rule S-MUPDATE is applied. As a consequence, $X_2 = \{x\}$ and $\mathcal{S}_2 = \mathcal{S}_3$. The rule MUPDATE$_h$ give us $Y_2 = \{x\}$ and the result follows. By application of the induction hypotheses, $(Y_1 \cup \{x\}) \subseteq (X_1 \cup \{x\})$.

• **Case Conditional** $\texttt{if}(e)\{s_{\texttt{true}}\}\{s_{\texttt{false}}\}$. Based on the rule S-IF, we have $\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc}^{X_1} \langle \mathcal{S}_2, \dot{v}\rangle \wedge (1)\langle \mathcal{S}_2, s_{\texttt{true}}\rangle \Rightarrow_{pc}^{X_2} \mathcal{S}_t \wedge (2)\langle \mathcal{S}_2, s_{\texttt{false}}\rangle \Rightarrow_{pc}^{X_3} \mathcal{S}_f$. Let $\mathcal{S}_b = \mathcal{S}_t \sqcup \mathcal{S}_f$.

The hybrid reduction is ruled by IF_h. We have to show:

$$\forall \mathcal{S}_0 \,.\, \langle \mathcal{S}_b \sqcup \mathcal{S}_0, e\rangle \rightarrow_{pc}^{Y_1} \langle \mathcal{S}_3, v^\sigma\rangle \wedge \langle \mathcal{S}_3, s_{\texttt{true}}\rangle \Rightarrow_{pc\sqcup\sigma}^{Y_2} \mathcal{S}_4 \wedge \langle \mathcal{S}_3, s_{\texttt{false}}\rangle \Rightarrow_{pc\sqcup\sigma}^{Y_3} \mathcal{S}_5 \\ \wedge \langle \mathcal{S}_4 \sqcup \mathcal{S}_5, s_v\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_6 \quad \Rightarrow (X_1 \cup X_2 \cup X_3) \subseteq (Y_1 \cup Y_2 \cup Y_3 \cup Y_4)$$

By application of the induction hypotheses, $Y_1 \subseteq X_1$, $Y_2 \subseteq X_2$, and $Y_3 \subseteq X_3$. We have two cases:

case $v = \texttt{true}$**:** In this case, $\langle \mathcal{S}_4 \sqcup \mathcal{S}_5, s_{\texttt{true}}\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_6$. By (1), it is possible to apply the induction hypotheses which give us $Y_4 \subseteq X_2$. Then, the result follows.

case $v = \texttt{false}$**:** In this case, $\langle \mathcal{S}_4 \sqcup \mathcal{S}_5, s_{\texttt{false}}\rangle \rightarrow_{pc\sqcup\sigma}^{Y_4} \mathcal{S}_6$. By (2), it is possible to apply the induction hypotheses which give us $Y_4 \subseteq X_3$. Then, the result follows.

• **Rest of the cases**. The following cases:
Case Variable x (ruled by S-VAR and VAR)
Case Literal l (ruled by S-LIT and LIT)
Case Binary Operation $e \oplus e$ (ruled by S-BIOP and BIOP)
Case Expression Statement e (ruled by S-EXPR and EXPR)
Case Null Statement `skip` (ruled by S-SKIP and SKIP)
Case Loop $\texttt{while}(e)\{s\}$ (ruled by S-WHILE and WHILE)
Case Sequence $s_1\,;s_2$ (ruled by S-SEQ and SEQ) are immediate by successive application of the induction hypothesis in each case.

Lemma 7 (The static part of $\mathcal{L}_h$ upgrades every written variable).

$$\forall s \in \mathcal{L}_h \,.\, \langle S_1, s\rangle \Rightarrow_{pc}^{X} S_2 \quad \Rightarrow pc \sqsubseteq \mathcal{S}_2[X]$$

Proof. By induction on the length of the execution. Following, the proof by case:

• **Case Static Assignment** $x = e$. Based on the rule S-ASSIGN we have to show

$$\langle \mathcal{S}_1, e\rangle \Rightarrow_{pc}^{X_1} \langle \mathcal{S}_2, \dot{v}\rangle \wedge \mathcal{S}_2[x \xleftarrow{\texttt{undef}^\perp} \dot{v}] \Downarrow_{pc}^{X_2} \mathcal{S}_3 \quad \Rightarrow pc \sqsubseteq \mathcal{S}_3[X_1 \cup X_2]$$

Let $\mathcal{S}_{2\texttt{undef}^\perp}(x) = \dot{w}$. By application of the induction hypotheses, $pc \sqsubseteq \mathcal{S}_2[X_1]$. We have two cases:

case $w = v$**:** The rule S-MUPDATEVS is applied. As a consequence, $X_2 = \emptyset$ and $\mathcal{S}_2 = \mathcal{S}_3$. The result $pc \sqsubseteq \mathcal{S}_3[X]$, where $X = X_1$, follows.

case $w \neq v$**:** The rule S-MUPDATE is applied. As a consequence, $X_2 = \{x\}$ and $\mathcal{S}_3 = \mathcal{S}_2[x \mapsto \dot{w}^{pc}]$. The result $pc \sqsubseteq \mathcal{S}_3[X]$, where $X = X_1 \cup \{x\}$, follows.

• **Rest of the cases**. The following cases:
Case Static Binary Operation $e \oplus e$ (ruled by S-BIOP)

Case Static Expression Statement e (ruled by S-Expr)
Case Static Conditional $\texttt{if(}e\texttt{)\{}s_{\texttt{true}}\texttt{\}\{}s_{\texttt{false}}\texttt{\}}$ (ruled by S-If)
Case Static Loop $\texttt{while(}e\texttt{)\{}s\texttt{\}}$ (ruled by S-While)
Case Static Sequence $s_1\texttt{;}s_2$ (ruled by S-Seq)
are immediate by successive application of the induction hypothesis in each case.

The following cases:
Case Static Variable x (ruled by S-Var)
Case Static Literal l (ruled by S-Lit)
Case Static Null Statement `skip` (ruled by S-Skip)
are immediate since, $X = \emptyset$ in each case.

Lemma 8 (Soundness of static MUpdate).

$$(1)\mathcal{M}_1 \overset{\lambda}{\sim}_\Delta \mathcal{M}'_1 \wedge \dot{v} \overset{\lambda}{\sim} \dot{v}' \wedge \mathcal{M}_1[x \xleftarrow{\Delta} \dot{v}]\Downarrow^{\sigma}_{pc} \mathcal{M}_2 \wedge \mathcal{M}'_1[x \xleftarrow{\Delta} \dot{v}']\Downarrow^{\sigma'}_{pc} \mathcal{M}'_2 \quad \Rightarrow \mathcal{M}_2 \overset{\lambda}{\sim}_\Delta \mathcal{M}'_2$$

Proof. Based on the rules

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

$$\text{S-MUpdateVS}\frac{\mathcal{M}_\Delta(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\Downarrow_\sigma \mathcal{M}}$$

Let $\mathcal{M}_\Delta(x) = w^\omega$ and $\mathcal{M}'_\Delta(x) = w'^{\omega'}$. By (1) and Lemma 1, (3) $w^\omega \overset{\lambda}{\sim} w'^{\omega'}$. We have the following two cases.

case $\omega \not\sqsubseteq \lambda$**:** So, $\omega' \not\sqsubseteq \lambda'$ by (3). The result follows immediately.

case $\omega \sqsubseteq \lambda$**:** So, $\omega' \sqsubseteq \lambda'$ by (3). Furthermore, $w = w'$. There are two possible subcases from here:

- **case** $w \neq v$**:** The rule S-MUpdate is applied to both updates. By (3) we know $w^{\sigma \sqcup \omega} \overset{\lambda}{\sim} w'^{\sigma' \sqcup \omega'}$. From here, the result $\mathcal{M}[x \mapsto w^{\sigma \sqcup \omega}] \overset{\lambda}{\sim}_\Delta \mathcal{M}'[x \mapsto w'^{\sigma' \sqcup \omega'}]$ follows.
- **case** $w = v$**:** The result follows directly from the assumption (1) after the application of S-MUpdateVS.

Lemma 9 (Soundness of MUpdate$_h$).

$$(1)\mathcal{M}_1 \overset{\lambda}{\sim}_\Delta \mathcal{M}'_1 \wedge (2)\dot{v} \overset{\lambda}{\sim} \dot{v}' \wedge \mathcal{M}_1[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}_2 \wedge \mathcal{M}'_1[x \xleftarrow{\Delta} \dot{v}']\downarrow_{pc} \mathcal{M}'_2 \quad \Rightarrow \mathcal{M}_2 \overset{\lambda}{\sim}_\Delta \mathcal{M}'_2$$

Proof. Based on the rules

$$\text{MUPDATE}_h \frac{\mathcal{M}_\Delta(x) = \dot{w} \quad w \neq v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}^{pc}]}$$

$$\text{MUPDATE-VS}_h \frac{\mathcal{M}_\Delta(x) = \dot{w} \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}}$$

Analogous to Lemma 8.

Definition 7 (Map join).

$$\mathcal{M}_1 \sqcup \mathcal{M}_2 \stackrel{\text{def}}{=\!=} \{x \mapsto v^{\sigma_1 \sqcup \sigma_2} \mid \mathcal{M}_1(x) = v^{\sigma_1} \wedge \mathcal{M}_2(x) = v^{\sigma_2}\}$$

Lemma 10 (λ-equivalence is a congruence with respect to join).

$$\begin{aligned} &\forall x \,.\, \exists v \mid M_1(x) = v^{\sigma_1} \wedge M_1'(x) = v^{\sigma_1'} \wedge \\ &\forall x \,.\, \exists v \mid M_2(x) = v^{\sigma_2} \wedge M_2'(x) = v^{\sigma_2'} \wedge \\ &\qquad {}_{(1)}\mathcal{M}_1 \overset{\lambda}{\sim} \mathcal{M}_1' \wedge {}_{(2)}\mathcal{M}_2 \overset{\lambda}{\sim} \mathcal{M}_2' \quad \Rightarrow \mathcal{M}_1 \sqcup \mathcal{M}_2 \overset{\lambda}{\sim} \mathcal{M}_1' \sqcup \mathcal{M}_2' \end{aligned}$$

Lemma 11. $\sigma \sqsubseteq S_1[X] \wedge \sigma \sqsubseteq S_2[X] \quad \Rightarrow \sigma \sqsubseteq (S_1 \sqcup \mathcal{S}_2)[X]$

C Permissiveness proof

In this appendix we prove that the value-sensitive variant of the existing monitors increments their permissiveness. This means that all the possible runs accepted by a value-insensitive monitor are also accepted by its value-sensitive alternative. In addition, to prove that the second strictly subsumes the former, it is enough to find a run in a program that is exclusively accepted by the value-sensitive variant.

This holds for all the presented enforcements. We formally prove permissiveness for the $\mathcal{L}$ case. All the other cases are analogous to this proof.

The Section C.1 considers $\mathcal{L}$ without the MUPDATE-VS rule. With only the rule MUPDATE to govern the side-effects, a non-sensitive upgrade [1] discipline is enforced.

C.1 $\mathcal{L}$ strictly more permissive than NSU

Theorem 13 proves that $\mathcal{L}$ is as permissive as NSU, i.e. $\mathcal{L}$ without MUPDATE-VS. The example in the Listing 1.1 shows that the subsuming is strict. The reductions of the value-insensitive variant of NSU are denote as $\dashrightarrow$.

Theorem 13 ($\mathcal{L}$ is as permissive as NSU).

$$\forall s \in \mathcal{L} \,.\, \langle \mathcal{S}_1, s\rangle \dashrightarrow_{pc} \mathcal{S}_2 \quad \Rightarrow \langle \mathcal{S}_1, s\rangle \rightarrow_{pc} \mathcal{S}_2$$

Proof. We have to show that every reduction $\dashrightarrow$ can be mimicked by a reduction $\rightarrow$. Let $\mathcal{R}_{\rightarrow}$ and $\mathcal{R}_{\dashrightarrow}$ the set of rules to reduce $\rightarrow$ and $\dashrightarrow$ respectively. We know that $\mathcal{R}_{\rightarrow} = \mathcal{R}_{\dashrightarrow} \cup \{\text{MUPDATE-VS}\}$. Therefore, $\mathcal{R}_{\dashrightarrow} \subset \mathcal{R}_{\rightarrow}$. Assume there exists a transition $\dashrightarrow$ such that $\rightarrow$ cannot follow. That would mean that there is a rule that exists in $\mathcal{R}_{\dashrightarrow}$ but not in $\mathcal{R}_{\rightarrow}$. This cannot happen, since $\mathcal{R}_{\dashrightarrow} \subset \mathcal{R}_{\rightarrow}$. Then, the result is proved by contradiction.

D Other variations of the value-sensitivity principle

In general, the value-sensitivity approach is orthogonal to existing dynamic and hybrid mechanisms for information flow tracking. Information flow enforcements track side effects to detect dependencies and update the labeling information of variables in runtime to reflect that dependency. Many suggested monitor in the pass can be described this way.

The value-sensitivity principle can be applied on top of these enforcements if the current value to relabel is considered. When the monitored side effects preserve the value invariant, the label does not have to be updated.

To illustrate the orthogonality of the value-sensitivity principle, we are going to consider to common variations of dynamic enforcements: a flow-insensitivity monitor and a hybrid monitor.

D.1 Flow-insensitivity

If a monitor does not allow security labels to change during the execution, it is called is flow-insensitive. Flow-insensitivity is analogous to strong typing. Once a variable is labeled (assigned to a type), its label (type) remains the same until the end of the execution.

Listing 1.8

```
h = 1^H;
l = 1^L;

l = h;
```

See the snippet in Listing 1.8. This program is diverge running under a traditional flow-insensitive monitor since the variable l cannot change its security label.

If we consider the flow-insensitive monitor from [26], the side effects are rules by the following FI-MUpdate. It is possible to apply the value-sensitivity approach to a flow-insensitive monitor by adding the rule FI-MUpdate-VS.

$$\text{FI-MUpdate}\frac{\mathcal{M}_\Delta(x) = w^\sigma \quad pc \sqsubseteq \sigma \quad v \sqsubseteq \sigma}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}[x \mapsto \dot{v}]}$$

$$\text{FI-MUpdate-VS}\frac{\mathcal{M}_\Delta(x) = w^\sigma \quad (pc \not\sqsubseteq \sigma \vee v \not\sqsubseteq \sigma) \quad w = v}{\mathcal{M}[x \xleftarrow{\Delta} \dot{v}]\downarrow_{pc} \mathcal{M}}$$

When the second rule is included, the execution of Listing 1.8 finishes with $l = 1^L$. The value of the target does not change and the variable upgrade can be safely ignored, resulting in an increment of permissiveness.