Permissive Dynamic Information Flow Analysis Technical Report #UCSC-SOE-09-34

Thomas H. Austin and Cormac Flanagan

University of California, Santa Cruz

Abstract. A key challenge in dynamic information flow analysis is handling *implicit flows*, where code conditional on a private variable updates a public variable `x`. The naive approach of upgrading `x` to private results in `x` being *partially-leaked*, where its value contains private data but its label may be either private (on this execution) or public (on an alternative execution where the conditional update was not performed).
Prior work proposed the *no-sensitive-upgrade* check, which handles implicit flows by prohibiting partially-leaked data, but attempts to update a public variable from a private context causes execution to get stuck.
To overcome this limitation, we develop a sound yet flexible *permissive-upgrade* strategy. To prevent information leaks, partially-leaked data is permitted but carefully tracked, and it must be *upgraded* to private before being used in a conditional test. We present an automatic dynamic analysis technique for inferring these upgrade annotations and inserting them into the program source code. The combination of these techniques allows more programs to run to completion, while still guaranteeing termination-insensitive non-interference in a purely-dynamic manner.

1 Introduction

The error-prone nature of software systems suggests that critical security policies are best enforced by small trusted modules, rather than being an emergent property of complex and buggy application code. In particular, the advent of memory-safe languages provides a resilient defense against notorious buffer-overrun vulnerabilities.

Applications in memory-safe languages are still vulnerable to other security problems, however, such as violations of privacy or data integrity expectations. Such security concerns are particularly relevant in a browser setting, where JavaScript code fragments from multiple untrusted or semi-trusted servers execute within the same process. Indeed, browsers are notoriously vulnerable to cross-site scripting attacks, which exploit confusions over the degree of authority or trust that should be granted to various code or data fragments. Tracking information flow and enforcing information security policies in the browser's JavaScript runtime engine provides a promising approach for addressing these kinds of higher-level security problems.

Much prior work has focused on type-based information flow analysis (1; 2), but these ideas are not applicable to dynamically-typed languages such as

JavaScript. Instead, our work focuses on enforcing information flow policies dynamically rather than statically. The central correctness property we wish to enforce is *termination-insensitive non-interference*, which says that changing the private inputs to an application should not influence any of the public outputs. Verifying this property dynamically requires simultaneously reasoning about the current execution of the program, as well as possible alternative executions of the program on the same public inputs but different private inputs.

The key challenge that we address is handling *implicit flows*, such as in the following code fragment, where code conditional on a private variable `x` updates a public variable `y`:

```
if (x) { y = false; }          // line 1
```

We assume that `y` is initially $\texttt{true}^L$, where the superscript L indicates public data with low confidentiality. Conversely, a superscript H indicates private data with high confidentiality. We use this code snippet to illustrate how our proposed approach improves over prior work.

Naive: A naive strategy for handling the above assignment is to upgrade the label on `y` to H, since that assignment is conditional on the private variable `x`. However, if `x` is $\texttt{true}^H$ then `y` becomes $\texttt{false}^H$; if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$. Thus, we say that the variable `y` is *partially-leaked*, since `y` now contains private information but `y` is labeled private on only *one* of these two executions.

Continuing the above example, suppose we now perform a second conditional assignment, where `z` is initially $\texttt{true}^L$.

```
if (y) { z = false; }          // line 2
```

Then the result of these two lines of code is that `z` is labeled public, but contains the value of the private input `x`. (That is, if `x` is $\texttt{true}^H$ then `y` becomes $\texttt{false}^H$ and `z` remains $\texttt{true}^L$; conversely, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$ and so `z` becomes $\texttt{false}^L$.) Thus, the naive approach to handling implicit flows permits both partially-leaked and totally-leaked data, and so does not provide termination-insensitive non-interference.

No-Sensitive-Upgrade: To remedy this limitation, prior work proposed the no-sensitive-upgrade check (3; 4), which prohibits partially-leaked data completely. Under this strategy, the above assignment to the public variable `y` from code conditional on a private variable `x` would get stuck. Although this strategy satisfies termination-insensitive non-interference,[1] stuck executions are undesirable since they violate expected liveness properties of the application, which motivates our development of a more flexible permissive-upgrade strategy.

[1] As in other approaches, the termination channel may leak one bit of data, or somewhat more in the presence of intermediary outputs (5).

Permissive-Upgrade: Our proposed permissive-upgrade strategy tolerates and carefully tracks partially-leaked data, while still providing termination-insensitive non-interference. Essentially, in addition to the private (H) and public (L) confidentiality labels, we introduce an additional label P to identify *partially-leaked* data that contains private information but which *may be labeled as public in some alternative executions*. Thus, at line 1 above, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$, as the assignment is not performed. If `x` is $\texttt{true}^H$ then `y` is updated to $\texttt{false}^P$ (where the label P reflects that in other executions `y` may remain labeled as public).

Such partially-leaked data must be handled quite delicately. In particular, if `y` is ever used in a conditional branch, as on line 2 above, then the permissive-upgrade strategy still gets stuck (now at line 2 rather than line 1), in order to avoid information leaks.

To avoid getting stuck, `y` can be upgraded to private before the conditional test, as in:

```
if (<H>y) { z = false; }          // line 2 alternate
```

This upgrade operation `<H>y` converts both public (L) and partially-leaked (P) data to private (H). Critically, upgrading partially-leaked data to private is sound since, as a consequence of the upgrade operation, the resulting data is labeled private on *all* executions, including alternative executions where `y` may originally have been labeled public. Thus, we can avoid stuck executions simply by inserting upgrade annotations at all sensitive uses of partially-leaked data.

To avoid the programmer overhead of manually-inserting annotations, we present an extension of our evaluation semantics that also infers these upgrade annotations. In situations where our original semantics would get stuck because of a sensitive use of partially-leaked data, the extended semantics implicitly inserts the appropriate upgrade annotation instead, and so continues execution. Thus, the conditional test "`if (y)`" at line 2 above is implicitly converted to "`if (<H>y)`".

In practice, we envision these techniques being applied as follows: A JavaScript web application is initially released in an instrumented form that uses the extended semantics to infer upgrade annotations. The extended semantics never gets stuck but does not (yet) provide information-flow guarantees. Once the set of dynamically-inferred annotations appears to converge (which must eventually happen, since the program is finite), the appropriately-annotated application could be re-released under the original permissive-upgrade semantics, with strong information-flow guarantees. Subsequently, some executions may still get stuck, but these are likely to be few, and can immediately be used to annotate the application, preventing subsequent executions from getting stuck at the same sensitive operation. In this manner, the difficulty of inferring upgrade annotations can be amortized over a large collection of users.

We hope that these annotation-inference techniques may help migrate existing Javascript web applications into a more secure world, where information flow policies are tracked and enforced by the language runtime itself. This deployment strategy does require information-flow support in the browser's JavaScript

implementation–in ongoing work with Mozilla, we are exploring how to incorporate such extensions in the Firefox browser (6).

Several topics remain for future work, including using a browser implementation to evaluate how these ideas scale to large web applications and what security policies are useful in practice. This paper provides a theoretical foundation for these later developments and implementations.

2 The λ^{info} Language

We formalize our permissive upgrade strategy in terms of λ^{info}, an imperative extension of the lambda calculus described in Figure 1. Terms include variables (x), constants (c), functions ($\lambda x.e$), and function application ($e_1\ e_2$). Since many of the challenges in information flow analysis come from imperative updates, our language supports mutable reference cells, including terms for allocating (`ref` e), dereferencing (`!`e), and updating (e_1 `:=` e_2) a reference cell. Finally, there is a term for labeling data as private ($\langle H \rangle e$).

This language is much simpler than full JavaScript, but we believe it allows us to deal with many of the essential complexities of implicit flows while minimizing syntactic clutter. We note that many additional constructs can be built from this core; the second part of Figure 1 sketches some standard encodings for booleans, conditionals, let-expressions, and sequential composition.

3 Three Evaluation Strategies for Implicit Flows

This section formalizes three evaluation strategies for handling implicit flows. Figure 2 presents the core semantics that is common to all evaluation strategies.

Each reference cell is allocated at an address a. A store σ maps addresses to values. A value v has the form r^k, which combines both an information flow label k and a *raw value* r. A raw value r is either a constant (c), an address (a), or a closure ($\lambda x.e, \theta$), which is a pair of a λ-expression and a substitution θ that maps variables to values. The core semantics includes both public (L) and private (H) labels, as well as the partially-leaked label (P), which is used exclusively by the permissive-upgrade semantics. Labels are ordered by $L \sqsubseteq H \sqsubseteq P$, reflecting the constraints on how correspondingly labeled data is used. We use $\sqcup$ to denote the corresponding join operation, and use $\emptyset$ to denote both the empty store and the empty substitution.

Figure 2 defines the semantics of λ^{info} via the big-step evaluation relation:

$$\sigma, \theta, e \Downarrow_{pc} \sigma', v$$

This relation evaluates an expression e in the context of a store σ, a substitution θ, and the current label pc of the program counter, and returns the resulting value v and the (possibly modified) store σ'. The program counter label $pc \in \{L, H\}$ reflects whether the execution of the current code is conditional on private data.

The rules defining this evaluation relation are mostly straightforward, with some notable subtleties on how labels are handled. In particular, we adopt the

Figure 1: The Source Language λ^{info}

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	`ref` e	reference allocation
	`!`e	dereference
	e`:=`e	assignment
	$\langle H \rangle e$	labeling operation
x, y, z		*Variable*
c		*Constant*

Standard encodings:

$$
\begin{array}{rcl}
\mathit{true} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x \\
\mathit{false} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \stackrel{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \stackrel{\text{def}}{=} & (\lambda x.e_2)\ e_1 \\
e_1\ ;\ e_2 & \stackrel{\text{def}}{=} & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, \quad x \notin FV(e_2)
\end{array}
$$

invariant that the label on the resulting value v is at least as secret as the program counter ($pc \sqsubseteq \mathit{label}(v)$). Thus, for example, the [CONST] rule evaluates a const c to the labeled value c^{pc}. The [FUN] rule evaluates a function $(\lambda x.e)$ to a closure $(\lambda x.e, \theta)^{pc}$ that captures the current substitution and that includes the program counter label. The [VAR] rule for a variable reference x extracts the corresponding value $\theta(x)$ from the environment and strengthens its label to be at least pc, using the following overloading of the join operator:

$$(r^l) \sqcup k \quad \stackrel{\text{def}}{=} \quad r^{(l \sqcup k)}$$

The [LABEL] rule for $\langle H \rangle e$ explicitly tags the result of evaluating e as private, ignoring the original label k. The [APP] rule applies a closure to an argument; to avoid information leaks, this rule gets stuck if the closure is partially-leaked. The [PRIM] rule applies function primitives. The [REF] and [DEREF] rules create and dereference a reference cell, respectively.

From these rules, we can derive corresponding evaluation rules for the encoded constructs, which are also shown in Figure 2. Critically, the [THEN] and [ELSE] rules get stuck if the conditional is partially-leaked.

Assignment statements are notably missing from Figure 2 since they introduce difficult problems with implicit flows. We present three strategies for tracking information flow across assignment statements, and illustrate these strategies on the example function `f(x)` shown in Figure 3. This function creates two public

Figure 2: Core Semantics for λ^{info}

Runtime Syntax:

$$\begin{array}{rcll} a & \in & \mathit{Address} & \\ \sigma & \in & \mathit{Store} & = \quad \mathit{Address} \rightarrow_p \mathit{Value} \\ \theta & \in & \mathit{Subst} & = \quad \mathit{Var} \rightarrow_p \mathit{Value} \\ r & \in & \mathit{RawValue} & ::= \quad c \mid a \mid (\lambda x.e, \theta) \\ v & \in & \mathit{Value} & ::= \quad r^k \\ k, l, pc & \in & \mathit{Label} & ::= \quad L \mid H \mid P \end{array}$$

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

[CONST]

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}}$$

[FUN]

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}}$$

[VAR]

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)}$$

[LABEL]

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', r^k}{\sigma, \theta, \langle H \rangle e \Downarrow_{pc} \sigma', r^H}$$

[APP]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v}$$

[PRIM]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, r^{k \sqcup l}}$$

[REF]

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow_{pc} \sigma', v \\ a \notin dom(\sigma')\end{array}}{\sigma, \theta, (\texttt{ref}\ e) \Downarrow_{pc} \sigma'[a := v], a^{pc}}$$

[DEREF]

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, \texttt{!}e \Downarrow_{pc} \sigma', (\sigma'(a) \sqcup k)}$$

Derived Evaluation Rules:

[THEN]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (true, \theta)^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \Downarrow_{pc} \sigma', v}$$

[ELSE]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (false, \theta)^k \\ k \neq P \\ \sigma_1, \theta, e_3 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \Downarrow_{pc} \sigma', v}$$

[LET]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \Downarrow_{pc} \sigma', v}$$

[SEQ]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v}$$

Figure 3: A Function With Problematic Implicit Flows

Function `f(x)`	`x=false`^{H}	`x=true`^{H}					
	All strategies	*Naive*		*NSU*		*Permissive U.*	
	`y` `z`	`y`	`z`	`y`	`z`	`y`	`z`
`let y = ref true in`							
`let z = ref true in`	`true`^{L} `true`^{L}	`true`^{L}	`true`^{L}	`true`^{L}	`true`^{L}	`true`^{L}	`true`^{L}
`if x then y:=false;`	`true`^{L} `true`^{L}	`false`^{H}	`true`^{L}	*stuck*		`false`^{P}	`true`^{L}
`if !y then z:=false;`	`true`^{L} `false`^{L}	`false`^{H}	`true`^{L}			*stuck*	
`!z`							
Return Value:	`false`^{L}	`true`^{L}					

reference cells `x` and `y` and conditionally updates both of them. If the argument `x` is `false`^{H}, then all three evaluation strategies proceed in the same manner by leaving `y` as `true`^{L} and updating `z` to `false`^{L}, and then returning `false`^{L}, as shown in the "*All strategies*" column. The following subsections describe how different strategies handle the tricky case where `x` is `true`^{H} and where `f` must update the public reference cell `y`.

3.1 The Naive Approach

The intuitive approach for assignment is to promote the label on the reference cell to at least the label k on the address a^k. (Note that a global evaluation invariant ensures that $pc \sqsubseteq k$.)

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad \text{[ASSIGN-NAIVE]}$$

For the function call `f(true`^{H}`)`, this strategy updates `y` to `false`^{H} but leaves `z` as `true`^{L}. Thus, by comparing the return value for the *All strategies* and *Naive* column of Figure 3, we see that the result of `f(x)` is a publicly-labeled copy of its private argument, and so this naive approach leaks information.

3.2 The No-Sensitive-Upgrade Approach

The no-sensitive-upgrade (NSU) approach avoids information leaks by getting stuck if a public reference cell is updated when the *pc* is private, or when the label on the target address is private. (In an implementation such stuck states might cause an exception to be thrown to the top level.)

The following rule requires that the label k on the target address a^k is not greater than the label on the reference cell contents. This rule assumes all data is labeled public or private, but never partially-leaked.

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ k \sqsubseteq \mathit{label}(\sigma_2(a))\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad \text{[ASSIGN-NSU]}$$

Figure 4: A Secure Function

Function `g(x)`	`x=false`^{H}	`x=true`^{H}	
	Both Strategies	*NSU*	*Permissive U.*
`let y = ref true in`	`true`^{L}	`true`^{L}	`true`^{L}
`if x then y := false;`	`true`^{L}	*stuck*	`false`^{P}
`y := true;`	`true`^{L}		`true`^{L}
`y`			
Return Value:	`true`^{L}		`true`^{L}

For our example function, the call `f(true`^{H}`)` would get stuck on the update for the public variable `z` within a private branch of execution, as illustrated by the NSU column of Figure 3, preventing the information leak.

Unfortunately, the NSU strategy may also get stuck on code that does not leak information, as shown in Figure 4. Although there is no information leak, evaluation of `g(true`^{H}`)` gets stuck when the private parameter `x` is partially-leaked. Thus, the NSU strategy satisfies termination-insensitive non-interference, but is unnecessarily restrictive.

3.3 The Permissive-Upgrade Approach

The permissive-upgrade semantics introduces an additional label (P) in order to tolerate and track partially-leaked data.

The rule [ASSIGN-PERMISSIVE] below considers an assignment to an address a^k that currently holds a value labelled l, and the rule uses the function $\mathit{lift}(k, l)$ defined below to infer the new label m for the reference cell. If execution is not in a private context ($k = L$), then there are no difficulties with implicit flows. Conversely, if we are updating a public reference cell ($l = L$) from a private context ($k = H$), then the rule labels the new contents as partially-leaked (P). Updating a private cell from a private context results in a private cell, and updating a partially-leaked cell from a private context leaves the cell as partially-leaked.

[ASSIGN-PERMISSIVE]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ k \neq P \\ l = \mathit{label}(\sigma_2(a)) \\ m = \mathit{lift}(k, l)\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup m)], v}$$

k	l	$\mathit{lift}(k, l)$
L	*any*	L
H	L	P
H	H	H
H	P	P

Figure 4 demonstrates that, under the permissive-upgrade strategy, the function `g` runs to completion on all boolean inputs (unlike under NSU). More generally, the following theorem shows that any execution that does not get stuck

under NSU evaluation (denoted $\Downarrow^{\text{nu}}_{pc}$) will also not get stuck under permissive upgrade evaluation (denoted $\Downarrow_{pc}$). Thus, the permissive upgrade strategy is strictly superior to NSU.

Theorem 1. *Suppose σ, θ, and pc do not contain the partially-leaked label P and $\sigma, \theta, e \Downarrow^{\text{nu}}_{pc} \sigma', v$. Then $\sigma, \theta, e \Downarrow_{pc} \sigma', v$, and σ' and v do not contain P.*

Partially-leaked data must be handled quite delicately, since on an alternative execution this data may be labeled as public. In particular, function calls, conditionals, and assignments are considered *sensitive* operations; these operations *get stuck* (via the antecedent $k \neq P$) if applied to partially-leaked data (as otherwise our information flow analysis could not track how alternative executions may propagate partially-leaked information). These stuck sensitive operations are critical for avoiding information leaks, and they distinguish the permissive-upgrade approach from the unsound naive approach.

For the function call `f(true`^{H}`)` from Figure 3, the permissive upgrade strategy handles the first conditional assignment by marking `y` as partially-leaked, but gets stuck on the second conditional test, to avoid information leaks.

We can remedy this situation by introducing the upgrade annotation `<H>`:

```
if <H>(!y) then z := false;
```

This upgrade annotation ensures the test expression is private on both executions, rather than partially-leaked on one execution and public on the other. The modified function `f` now runs to completion on all boolean inputs. Section 6 discusses how to infer these upgrade annotations automatically.

The following function `h(x)` clarifies that assignment statements must also be considered sensitive operations. The evaluation of `h(false`^{H}`)` returns `false`^{L}, whereas `h(true`^{H}`)` gets stuck at the assignment to `(!w)` in order to avoid returning `true`^{L} and thereby leaking information.

```
h  =def  λx. let y = ref true in
             let z = ref true in
             let w = ref y in
               if x then w := z;
               (!w) := false;
               !y
```

Adding the upgrade "`<H>(!w) := false`" allows both calls to complete without information leaks: `h(false`^{H}`)` returns `false`^{P} and `h(true`^{H}`)` returns `true`^{L}.

4 Termination-Insensitive Non-Interference

We now verify that the permissive-upgrade strategy guarantees termination-insensitive non-interference. The traditional non-interference argument is based on an equivalence relation between states that is transitive. However, the introduction of partially-leaked data in our semantics significantly complicates this proof, since the values `true`^{L} and `false`^{P} are considered equivalent, as are

$\texttt{false}^P$ and $\texttt{false}^L$, but $\texttt{true}^L$ and $\texttt{false}^L$ are not equivalent. Thus, the desired "equivalence" relation is no longer transitive, and so we call it a *compatibility* relation ($\sim$) instead. Intuitively, two stores are compatible if they differ only on private data, and executions that start with compatible stores should yield compatible results. In more detail, we define the compatibility relation ($\sim$) on labels, values, substitutions, and stores as follows.

- Two labels are *compatible* if both are private or one is partially-leaked:

  $$k_1 \sim k_2 \quad \stackrel{\text{def}}{=} \quad (k_1, k_2) \in \{(H,H), (P,-), (-,P)\}$$

  Label compatibility is neither reflexive (as $L \not\sim L$) nor transitive (as $L \sim P \sim L$ but $L \not\sim L$).
- Two values are compatible if either their labels are compatible or the labels are identical and the raw values are compatible.

  $$r_1^{k_1} \sim r_2^{k_2} \quad \stackrel{\text{def}}{=} \quad k_1 \sim k_2 \vee (k_1 = k_2 \wedge r_1 \sim r_2)$$
- Two raw values are compatible if they are identical or they are both closures with identical code and compatible substitutions:

  $$r_1 \sim r_2 \quad \stackrel{\text{def}}{=} \quad r_1 = r_2 \ \vee (r_1 = (\lambda x.e, \theta_1) \wedge r_2 = (\lambda x.e, \theta_2) \wedge \theta_1 \sim \theta_2)$$
- Two substitutions are compatible (written $\theta_1 \sim \theta_2$) if they have the same domain and compatible values:

  $$\theta_1 \sim \theta_2 \quad \stackrel{\text{def}}{=} \quad dom(\theta_1) = dom(\theta_2) \wedge \forall x \in dom(\theta_1).\ (\theta_1(x) \sim \theta_2(x))$$
- Two stores σ_1 and σ_2 are compatible (written $\sigma_1 \sim \sigma_2$) if they are compatible at all common addresses:

  $$\sigma_1 \sim \sigma_2 \quad \stackrel{\text{def}}{=} \quad \forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \sigma_1(a) \sim \sigma_2(a)$$

We also introduce an *evolution* (or *can evolve to*) relation ($\rightsquigarrow$) that constrains how evaluation with a private program counter can update the store. This relation composes in a transitive manner with compatibility: see Lemma 2 below.

- Label k_1 *can evolve to* k_2 if both labels are private or k_2 is partially-leaked:

  $$k_1 \rightsquigarrow k_2 \quad \stackrel{\text{def}}{=} \quad k_1 = k_2 = H \ \vee \ k_2 = P$$
- A value $r_1^{k_1}$ *can evolve to* $r_2^{k_2}$ if either the two values are equal or k_1 can evolve to k_2:

  $$r_1^{k_1} \rightsquigarrow r_2^{k_2} \quad \stackrel{\text{def}}{=} \quad r_1^{k_1} = r_2^{k_2} \quad \vee \quad k_1 \rightsquigarrow k_2$$
- A store σ_1 *can evolve* to σ_2 if every value in σ_1 can evolve to the corresponding value in σ_2:

  $$\sigma_1 \rightsquigarrow \sigma_2 \quad \stackrel{\text{def}}{=} \quad dom(\sigma_1) \subseteq dom(\sigma_2) \ \wedge \ \forall a \in dom(\sigma_1).\ \sigma_1(a) \rightsquigarrow \sigma_2(a)$$

The evolution relation captures how evaluation with a private program counter can update the store.

Lemma 1 (Evaluation Preserves Evolution).
If $\sigma, \theta, e \Downarrow_H \sigma', v$ *then* $\sigma \rightsquigarrow \sigma'$.

PROOF SKETCH: By induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$. The most interesting case is for [ASSIGN-PERMISSIVE], which introduces the label P to track partially-leaked data. □

If two stores are compatible ($\sigma_1 \sim \sigma_2$), then evolution of one store ($\sigma_2 \rightsquigarrow \sigma_3$) results in a new store that is compatible to the original stores ($\sigma_1 \sim \sigma_3$), with the caveat that any newly allocated address must not be in the original stores.

Lemma 2 (Evolution Preserves Compatibility of Stores).
If $\sigma_1 \sim \sigma_2 \rightsquigarrow \sigma_3$ *and* $(dom(\sigma_1) \setminus dom(\sigma_2)) \cap dom(\sigma_3) = \emptyset$ *then* $\sigma_1 \sim \sigma_3$.

Finally, we prove our central result: if an expression e is executed twice from compatible stores and compatible substitutions, then both executions will yield compatible resulting stores and values. That is, private inputs never leak into public outputs.

Theorem 2 (Termination-Insensitive Non-Interference).
Suppose $pc \in \{L, H\}$ *and* $\sigma_1 \sim \sigma_2$ *and* $\theta_1 \sim \theta_2$ *and* $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i$ *for* $i \in 1, 2$.
Then $\sigma_1' \sim \sigma_2'$ *and* $v_1 \sim v_2$.

PROOF SKETCH: By induction on the derivation of $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$. In the [APP] rule case, where the callee may be private, Lemma 2 above is critical for limiting the effect of this conditionally-executed code. Complete proofs are available in the appendix. □

5 Upgrade Inference

The permissive-upgrade semantics guarantees non-interference while getting stuck on fewer programs than the NSU semantics, and it will not get stuck if the program includes upgrade annotations on sensitive uses of partially-leaked data.

We now extend our semantics to infer these upgrade annotations. We begin by adding a position marker $p \in Position$ on each sensitive operation (applications and assignments) where partially-leaked data is not permitted.

$$e \ ::= \ \ldots \mid (e_1 \ e_2)^p \mid (e_1 := e_2)^p$$

Rather than explicitly insert upgrade annotations at particular positions in the source code, we instead extend the store σ to now also record the positions where these upgrades have been *conceptually* inserted.

We replace the original [APP] evaluation rule with three variants, and similarly for [ASSIGN-PERMISSIVE], as shown if Figure 5. The [APP-NORMAL] rule applies if an upgrade has not been inserted for this operation ($p \notin \sigma$) and is not needed ($k \neq P$). [APP-UPGRADE] handles situations where the upgrade has been inserted ($p \in \sigma$) by ignoring the label k on the closure and behaving as if the closure were labeled private instead. [APP-INFER] handles situations where an upgrade is required ($k = P$) but has not yet been inserted ($p \notin \sigma$); it adds this position tag to the store (conceptually inserting the required upgrade) and then reevaluates the application.

Figure 5: Upgrade Inference

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

[APP-NORMAL]
$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-NORMAL]
$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(k, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v}$$

[APP-UPGRADE]
$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_H \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-UPGRADE]
$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(H, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v}$$

[APP-INFER]
$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-INFER]
$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma', v}$$

Our revised semantics still guarantees non-interference, but only if the evaluation did not infer additional upgrades. This observation leads to some interesting design decisions. If output of the final result is allowed even when there was an inferred upgrade, then termination-insensitive non-interference is not guaranteed, but the information leak is detected. If output is forbidden in this case, then the behavior is identical to the permissive-upgrade semantics.

Theorem 3 (Non-Interference Of Upgrade Inference). *Suppose* $pc \neq P$ *and* $\sigma_1 \sim \sigma_2$ *and* $\theta_1 \sim \theta_2$ *and* $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i$ *and* $P_i = (\sigma_i' \setminus \sigma_i) \cap \mathit{Position}$ *for* $i \in 1, 2$. *If* $P_1 = P_2 = \emptyset$ *then* $\sigma_1' \sim \sigma_2'$ *and* $v_1 \sim v_2$.

We next show that adding some upgrades A to a program only influences the labels in the program's result, but not the raw values.

To formalize this property, we introduce a *raw equivalence* order ($\approx$) that identifies values, substitutions, and stores that differ *only* in their labels, not in their underlying raw values. Moreover, raw equivalent stores are allowed to differ in the position tags that they include, *i.e.*, $\sigma \approx (\sigma \cup A)$.

Theorem 4 (Non-Interference Of Upgrade Annotations). *Suppose* $pc \neq P$ *and* $A \subseteq \mathit{Position}$ *and* $\sigma, \theta, e \Downarrow_{pc} \sigma_1, v_1$ *and* $(\sigma \cup A), \theta, e \Downarrow_{pc} \sigma_2, v_2$. *Then* $\sigma_1 \approx \sigma_2$ *and* $v_1 \approx v_2$

We prove this theorem via the following lemma, which strengthens the inductive hypothesis.

Lemma 3. *Suppose* $pc \neq P$ *and* $\sigma_1 \approx \sigma_2$ *and* $\theta_1 \approx \theta_2$ *and* $\sigma_i, \theta_i, e \Downarrow_{pc_i} \sigma_i', v_i$ *for* $i \in 1, 2$. *Then* $\sigma_1' \approx \sigma_2'$ *and* $v_1 \approx v_2$.

6 Related Work

Fenton's paper on memoryless subsystems (7) is largely the beginning of information flow analysis. Denning's papers (8; 9) highlight the challenges associated with implicit flows, and advocate a static certification approach; since then, static approaches have dominated because of their generally superior performance and the perceived advantages in handling implicit flows.

Volpano et al. (1) and Heintze and Riecke (2) are two of the most well known type-based approaches, though their target languages are relatively minimal. Pottier and Simonet (10) introduce a more complex system for Core ML.

Dynamic approaches have been applied mostly to integrity problems, including taint analysis for Perl, Ruby, and PHP. Integrity and confidentiality are usually claimed to be dual problems, but this is disputed. Sabelfeld and Myers (11) note that integrity can be damaged by a system error without any outside influence. Haack et al. (12) observe that since *format integrity errors* are unaffected by implicit flows, integrity analysis has focused on by dynamic techniques.

Recently, there has been more appreciation of the complementary benefits that each approach offers. Many strategies rely primarily on static techniques and insert dynamic runtime checks only in ambiguous cases (13; 14). This approach reduces false positives with a minimum impact on performance. Myers (15) introduced JFlow, a variant of Java using this hybrid strategy, which was the basis for Jif (16). Chugh et al. (17) propose a mostly static approach for analyzing JavaScript with "holes" for dynamically generated code.

Generally, dynamic analysis is more often applied to client-side scripting, particularly for JavaScript, where dynamic typing makes type-based approaches difficult, and the flexibility of the language makes offline certification ineffective. Vogt et al. (18) reverse the standard hybrid approach, relying primarily on dynamic checks but falling back to runtime certification for implicit flows. Russo et al. study information flow analysis in the DOM (19) and timeout mechanisms (20)–both major issues for JavaScript applications. Askarov and Sabelfeld (21) cover declassification and analysis of dynamic code evaluation.

In his dissertation, Zdancewic (3) first proposed rules for dynamic analysis to effectively handle implicit flows. Our own work later dubbed the key assignment rule the *no-sensitive-upgrade* check and addressed performance concerns for dynamic analysis with a sparse-labeling approach (4). Le Guernic et al. (22) use dynamic automaton-based monitoring. Sabelfeld and Russo (23) formally prove that both static and dynamic approaches make the same security guarantees.

Flow-sensitive approaches attempt to reduce false-positives in static analysis. Hunt and Sands (24) use a type-system to guarantee this property. Hammer and Snelting (25) use *program dependency graphs* to analyze in JVM bytecode.

Both Chong and Myers (26) and Fournet and Rezk (27) focus on downgrading confidential information. Askarov et al. (5) demonstrate that Denning-style analysis may leak more than one bit in the presence of intermediary output channels, but that any attack will be limited to a brute-force approach. Askarov and Sabelfeld (28) and King et al. (29) discuss exception handling challenges.

7 Conclusion

We present a permissive-upgrade semantics that tracks information flow in a more flexible manner than prior dynamic approaches, using a new label (P) to permit partially-leaked data without loss of soundness. To avoid stuck executions, upgrade annotations are required on sensitive uses of partially-leaked data, and we show how these upgrade annotations can be inferred dynamically. We hope these techniques will help enforce important information-flow policies in dynamically-typed web applications. In ongoing work with Mozilla (6), we are exploring how to incorporate these and other ideas into the Firefox web browser.

References

[1] Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2-3) (1996) 167–187

[2] Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: Symposium on Principles of Programming Languages. (1998) 365–377

[3] Zdancewic, S.A.: Programming languages for information security. PhD thesis, Cornell University, Ithaca, NY, USA (2002) Chair-Myers,, Andrew.

[4] Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, New York, NY, USA, ACM (2009) 113–124

[5] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security, Berlin, Heidelberg, Springer-Verlag (2008) 333–348

[6] Eich, B.: Mozilla FlowSafe: Information flow security for the browser https://wiki.mozilla.org/FlowSafe, accessed October 2009.

[7] Fenton, J.S.: Memoryless subsystems. The Computer Journal **17**(2) (1974) 143–147

[8] Denning, D.E.: A lattice model of secure information flow. Communications of the ACM **19**(5) (1976) 236–243

[9] Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(7) (1977) 504–513

[10] Pottier, F., Simonet, V.: Information flow inference for ML. Transactions on Programming Languages and Systems **25**(1) (2003) 117–158

[11] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on **21**(1) (Jan 2003) 5–19

[12] Haack, C., Poll, E., Schubert, A.: Explicit information flow properties in JML. In: 3rd Benelux Workshop on Information and System Security (WISSec). (2008)

[13] Chandra, D., Franz, M.: Fine-grained information flow analysis and enforcement in a java virtual machine. (Dec. 2007) 463–475
[14] Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: Information and Communications Security. (2006) 332–351
[15] Myers, A.C.: Jflow: Practical mostly-static information flow control. In: Symposium on Principles of Programming Languages. (1999) 228–241
[16] : Jif homepage http://www.cs.cornell.edu/jif/, accessed October 2009.
[17] Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for javascript. In: PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2009) 50–62
[18] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. (February 2007)
[19] Russo, A., Sabelfeld, A., Chudnov, A.: Tracking information flow in dynamic tree structures. In Backes, M., Ning, P., eds.: ESORICS. Volume 5789 of Lecture Notes in Computer Science., Springer (2009) 86–103
[20] Russo, A., Sabelfeld, A.: Securing timeout instructions in web applications. In: IEEE Computer Security Foundations Workshop. (2009)
[21] Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: IEEE Computer Security Foundations Workshop. (2009)
[22] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based confidentiality monitoring. (2006)
[23] Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Perspectives of System Informatics. (2009)
[24] Hunt, S., Sands, D.: On flow-sensitive security types. In Morrisett, J.G., Jones, S.L.P., eds.: POPL, ACM (2006) 79–90
[25] Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security
[26] Chong, S., Myers, A.C.: Security policies for downgrading. In: CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, New York, NY, USA, ACM (2004) 198–209
[27] Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: Symposium on Principles of Programming Languages. (2008) 323–335
[28] Askarov, A., Sabelfeld, A.: Catch me if you can: permissive yet secure error handling. In: PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, New York, NY, USA, ACM (2009) 45–57
[29] King, D., Hicks, B., Hicks, M., Jaeger, T.: Implicit flows: Can't live with 'em, can't live without 'em. In: International Conference on Information Systems Security. (2008) 56–70
[30] Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a java-like language. In: IEEE Computer Security Foundations Workshop, IEEE Computer Society (2002) 253–267

A Proofs

We first observe certain properties of labels. First of all, if two labels are compatible, then joining any other labels to either or both of the original labels will still result in compatible labels.

Lemma 4. *If* $k_1 \sim k_2$ *then* $(l_1 \sqcup k_1) \sim (l_2 \sqcup k_2)$.

Also, if two labels are compatible and are part of different values, those values will also be compatible.

Lemma 5. *If* $k_1 \sim k_2$ *then* $(v_1 \sqcup k_1) \sim (v_2 \sqcup k_2)$.

Finally, in a secure context (H as the first argument to the *lift* function), all labels are compatible.

Lemma 6. $\mathit{lift}(H, l_1) \sim \mathit{lift}(H, l_2)$.

We now proceed to prove Theorem 1.

Restatement of Theorem 1. Suppose σ, θ, and pc do not contain the partially-leaked label P and $\sigma, \theta, e \Downarrow_{pc}^{\text{nu}} \sigma', v$. Then $\sigma, \theta, e \Downarrow_{pc} \sigma', v$, and σ' and v do not contain P.

Proof. The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_{pc}^{\text{nu}} \sigma', v$ and by case analysis on the last rule used.

- The [CONST], [FUN], [VAR], [LABEL], [APP], [PRIM], [REF], and [DEREF] rules are identical for both semantics, and none of these rules produces the label P. Therefore, these cases hold by induction.
- In the [ASSIGN-NSU] case, $e = (e_1 := e_2)$, and from the antecedents of this rule, we have:

$$\begin{gathered}\sigma, \theta, e_1 \Downarrow_{pc}^{\text{nu}} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc}^{\text{nu}} \sigma_2, v \\ k \sqsubseteq \mathit{label}(\sigma_(a)) \\ \sigma' = \sigma_2[a := (v \sqcup k)]\end{gathered}$$

  By induction:

$$\begin{gathered}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v\end{gathered}$$

  Let $l = \mathit{label}(\sigma_2(a))$ and $m = \mathit{lift}(k, l)$. Then by [ASSIGN-PERMISSIVE], $\sigma, \theta, e_1 := e_2 \Downarrow_{pc} \sigma_2[a := (v \sqcup m)], v$. It remains to show that $\sigma_2[a := (v \sqcup m)] = \sigma'$, i.e. that $v \sqcup m = v \sqcup k$. We proceed by a case analysis on k:
  - case $k = P$: This cannot happen, since no evaluation rule in the NSU semantics produces P.
  - case $k = L$: Then $k \sqsubseteq \mathit{label}(\sigma_2(a))$. Also, $m = \mathit{lift}(k, \mathit{label}(\sigma_2(a))) = L$. Therefore by Lemma 5, $v \sqcup m = v \sqcup k$.
  - case $k = H$: By case analysis on $l = \mathit{label}(\sigma_2(a))$:

* case $l = P$: This cannot happen, since the initial store was free of P-labeled data, and no evaluation rule introduces the label P in the NSU semantics.
* case $l = L$: Then $k \not\sqsubseteq l$, so the NSU semantics is stuck.
* case $l = H$: Then $k \sqsubseteq l$ and $m = \mathit{lift}(k, l) = H$. Therefore by Lemma 5, $v \sqcup m = v \sqcup k$.

□

In order to prove the desired relationship between the evaluation and evolution relations, we first prove some preliminary lemmas.

Lemma 7. $\forall m.\ m \rightsquigarrow \mathit{lift}(H, m)$.

We note that the evolution relation is transitive, and that it is reflexive for both values and stores.

Lemma 8. $\rightsquigarrow$ *is transitive.*

Lemma 9. $\rightsquigarrow$ *on values and stores is reflexive.*

The evolution relation on values interacts in a "transitive" manner with the compatibility relation.

Lemma 10. *If* $v_1 \sim v_2 \rightsquigarrow v_3$ *then* $v_1 \sim v_3$.

Proof. If $v_2 = v_3$ then the lemma trivially holds. Otherwise let $v_i = r_i^{k_i}$ and consider the possibilities for $k_2 \rightsquigarrow k_3$.

- Suppose $k_2 = k_3 = H$. Then $k_1 \in \{H, P\}$ and so $k_1 \sim k_3$.
- Suppose $k_3 = P$. Then $k_1 \sim k_3$.

□

We now proceed to prove Lemma 1 and Lemma 2.

Restatement of Lemma 1 (Evaluation Preserves Evolution).
If $\sigma, \theta, e \Downarrow_H \sigma', v$ then $\sigma \rightsquigarrow \sigma'$.

Proof. The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$ and by case analysis on the final rule in the derivation.

- [CONST], [FUN], [VAR]: $\sigma' = \sigma$.
- [APP], [PRIM], [LABEL], [DEREF]: By induction.
- [REF]: σ and σ' agree on their common domain.
- [ASSIGN-PERMISSIVE]: In this case, $e = (e_1 := e_2)$ and we have:

$$\begin{gathered}\sigma, \theta, e_1 \Downarrow_H \sigma_1, a^H \\ \sigma_1, \theta, e_2 \Downarrow_H \sigma_2, v \\ l = \mathit{label}(\sigma_2(a)) \\ m = \mathit{lift}(H, l) \\ \sigma' = \sigma_2[a := (v \sqcup m)]\end{gathered}$$

By induction, $\sigma \rightsquigarrow \sigma_1 \rightsquigarrow \sigma_2$. By Lemma 7, $l \rightsquigarrow m$. Hence $\sigma_2(a) \rightsquigarrow (v \sqcup m)$ and so $\sigma_2 \rightsquigarrow \sigma'$.

□

Restatement of Lemma 2 (Evolution Preserves Compatibility of Stores). If $\sigma_1 \sim \sigma_2 \rightsquigarrow \sigma_3$ and $(dom(\sigma_1) \setminus dom(\sigma_2)) \cap dom(\sigma_3) = \emptyset$ then $\sigma_1 \sim \sigma_3$.

Proof. Let $D = dom(\sigma_1) \cap dom(\sigma_3)$. Then $D \sqsubseteq dom(\sigma_2)$. This means that $\forall a \in D.\ \sigma_1(a) \sim \sigma_2(a)$ and $\sigma_2(a) \rightsquigarrow \sigma_3(a)$. Therefore, by Lemma 10:

$$\forall a \in D.\ \sigma_1(a) \sim \sigma_3(a)$$

Hence by the definition of the evolution relation, $\sigma_1 \sim \sigma_3$. □

Restatement of Theorem 2 (Termination-Insensitive Non-Interference). Suppose $pc \in \{L, H\}$ and $\sigma_1 \sim \sigma_2$ and $\theta_1 \sim \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i$ for $i \in 1, 2$. Then $\sigma_1' \sim \sigma_2'$ and $v_1 \sim v_2$.

Proof. The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c^{pc}$.
- [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = (\theta_1(x) \sqcup pc) \sim (\theta_2(x) \sqcup pc) = v_2$.
- [FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = (\lambda x.e', \theta_1)^{pc} \sim (\lambda x.e', \theta_2)^{pc} = v_2$.
- [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

  $$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i', r_i^{k_i}$$

  By induction, $\sigma_1' \sim \sigma_2'$. Also, regardless of the raw values r_1 and r_2, $r_1^H \sim r_2^H$ by the definition of the compatibility relation.
- [APP]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

  $$\begin{gathered} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ k_i \neq P \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i' \\ \sigma_i''', \theta_i'[x := v_i'], e_i \Downarrow_{k_i} \sigma_i', v_i \end{gathered}$$

  By induction:

  $$\begin{gathered} \sigma_1'' \sim \sigma_2'' \\ \sigma_1''' \sim \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \sim (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \sim v_2' \end{gathered}$$

- If k_1 and k_2 are both H then $v_1 \sim v_2$, since they both have label at least H. By Lemma 1, $\sigma_i''' \rightsquigarrow \sigma_i'$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[2] *i.e.*:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{3-i}') = \emptyset$$

  Under this assumption, by Lemma 2 $\sigma_1''' \sim \sigma_2'$. Applying Lemma 2 again gives $\sigma_1' \sim \sigma_2'$.
- Otherwise $\theta_1' \sim \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \sim \sigma_2'$ and $v_1'' \sim v_2''$, and hence $v_1' \sim v_2'$.

– [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', c_i^{k_i} \\ \sigma_i'', \theta_i, e_a \Downarrow_{pc} \sigma_i', d_i^{l_i} \\ r_i = [\![c_i]\!](d_i)\end{gathered}$$

  By induction:

$$\begin{array}{ll}\sigma_1'' \sim \sigma_2'' & \sigma_1' \sim \sigma_2' \\ c_1^{k_1} \sim c_2^{k_2} & d_1^{l_1} \sim d_2^{l_2}\end{array}$$

  - If either $k_1 \sim k_2$ or $l_1 \sim l_2$, then by Lemma 4 $k_1 \sqcup l_1 \sim k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.
  - Otherwise, $r_1 = r_2$, since $c_1 = c_2$ and $d_1 = d_2$. Also, $k_1 \sqcup l_1 = k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

– [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $a^{pc} = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i'', v_i' \\ \sigma_i' = \sigma_i''[a := v_i']\end{gathered}$$

  By induction, $\sigma_1'' \sim \sigma_2''$ and $v_1' \sim v_2'$, and so $\sigma_1' \sim \sigma_2'$.

– [DEREF]: In this case, $e = \texttt{!}e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i', a_i^{k_i} \\ v_i = \sigma_i'(a_i) \sqcup k_i\end{gathered}$$

  By induction, $\sigma_1' \sim \sigma_2'$ and $a_1^{k_1} \sim a_2^{k_2}$.

  - Suppose $a_1^{k_1} = a_2^{k_2}$. Then $a_1 = a_2$ and $k_1 = k_2$ and $\sigma_1'(a_1) \sim \sigma_2'(a_2)$, and so $v_1 \sim v_2$.
  - Suppose $a_1^{k_1} \neq a_2^{k_2}$. Then since $a_1^{k_1} \sim a_2^{k_2}$ we must have that $k_1 \sim k_2$ and hence $v_1 \sim v_2$ from Lemma 5.

[2] We refer the interested reader to (30) for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.

– [ASSIGN-PERMISSIVE] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', a_i^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i \\ k_i \neq P \\ m_i = \mathit{lift}(k_i, \mathit{label}(\sigma_i'''(a_i))) \\ \sigma_i' = \sigma_i'''[a_i := v_i \sqcup m_i]\end{gathered}$$

By induction:

$$\begin{array}{ll}\sigma_1'' \sim \sigma_2'' & \sigma_1''' \sim \sigma_2''' \\ a_1^{k_1} \sim a_2^{k_2} & v_1 \sim v_2\end{array}$$

- If $k_1 \sim k_2$ then $k_1 = k_2 = H$. By Lemma 6, $m_1 \sim m_2$. By Lemma 5, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma_1' \sim \sigma_2'$.
- Otherwise $k_1 = k_2 = L$. Then $m_1 = m_2 = L$ and hence $\sigma_1' \sim \sigma_2'$.

□

Restatement of Theorem 3 (Non-Interference Of Upgrade Inference). Suppose $pc \neq P$ and $\sigma_1 \sim \sigma_2$ and $\theta_1 \sim \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i$ and $P_i = (\sigma_i' \setminus \sigma_i) \cap \mathit{Position}$ for $i \in 1, 2$. If $P_1 = P_2 = \emptyset$ then $\sigma_1' \sim \sigma_2'$ and $v_1 \sim v_2$.

Proof. The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

– [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c^{pc}$.
– [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = (\theta_1(x) \sqcup pc) \sim (\theta_2(x) \sqcup pc) = v_2$.
– [FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = (\lambda x.e', \theta_1)^{pc} \sim (\lambda x.e', \theta_2)^{pc} = v_2$.
– [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i', r_i^{k_i}$$

By induction, $\sigma_1' \sim \sigma_2'$. Also, regardless of the raw values r_1 and r_2, $r_1^H \sim r_2^H$ by the definition of the compatibility relation.
– [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', c_i^{k_i} \\ \sigma_i'', \theta_i, e_a \Downarrow_{pc} \sigma_i', d_i^{l_i} \\ r_i = [\![c_i]\!](d_i)\end{gathered}$$

By induction:

$$\begin{array}{ll}\sigma_1'' \sim \sigma_2'' & \sigma_1' \sim \sigma_2' \\ c_1^{k_1} \sim c_2^{k_2} & d_1^{l_1} \sim d_2^{l_2}\end{array}$$

- If either $k_1 \sim k_2$ or $l_1 \sim l_2$, then by Lemma 4 $k_1 \sqcup l_1 \sim k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.
- Otherwise, $r_1 = r_2$, since $c_1 = c_2$ and $d_1 = d_2$. Also, $k_1 \sqcup l_1 = k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

– [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $a^{pc} = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i'', v_i'$$
$$\sigma_i' = \sigma_i''[a := v_i']$$

By induction, $\sigma_1'' \sim \sigma_2''$ and $v_1' \sim v_2'$, and so $\sigma_1' \sim \sigma_2'$.

– [DEREF]: In this case, $e = \texttt{!}e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i', a_i^{k_i}$$
$$v_i = \sigma_i'(a_i) \sqcup k_i$$

By induction, $\sigma_1' \sim \sigma_2'$ and $a_1^{k_1} \sim a_2^{k_2}$.

- Suppose $a_1^{k_1} = a_2^{k_2}$. Then $a_1 = a_2$ and $k_1 = k_2$ and $\sigma_1'(a_1) \sim \sigma_2'(a_2)$, and so $v_1 \sim v_2$.
- Suppose $a_1^{k_1} \neq a_2^{k_2}$. Then since $a_1^{k_1} \sim a_2^{k_2}$ we must have that $k_1 \sim k_2$ and hence $v_1 \sim v_2$ from Lemma 5.

– [APP-NORMAL]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$p \notin \sigma_i$$
$$\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i}$$
$$k_i \neq P$$
$$\sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i'$$
$$\sigma_i''', \theta_i'[x := v_i'], e_i \Downarrow_{k_i} \sigma_i', v_i$$

By induction:

$$\sigma_1'' \sim \sigma_2''$$
$$\sigma_1''' \sim \sigma_2'''$$
$$(\lambda x.e_1, \theta_1')^{k_1} \sim (\lambda x.e_2, \theta_2')^{k_2}$$
$$v_1' \sim v_2'$$

- If k_1 and k_2 are both H then $v_1 \sim v_2$, since they both have label at least H. By Lemma 1, $\sigma_i''' \rightsquigarrow \sigma_i'$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space. *i.e.*:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{3-i}') = \emptyset$$

Under this assumption, by Lemma 2 $\sigma_1''' \sim \sigma_2'$. Applying Lemma 2 again gives $\sigma_1' \sim \sigma_2'$.

- Otherwise $\theta_1' \sim \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \sim \sigma_2'$ and $v_1'' \sim v_2''$, and hence $v_1' \sim v_2'$.

– [APP-UPGRADE]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \in \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i' \\ \sigma_i''', \theta_i'[x := v_i'], e_i \Downarrow_H \sigma_i', v_i \end{gathered}$$

By induction:

$$\begin{gathered} \sigma_1'' \sim \sigma_2'' \\ \sigma_1''' \sim \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \sim (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \sim v_2' \end{gathered}$$

By the final antecedent of the rule, both v_1 and v_2 must have a label at least H, so $v_i \sim v_s$. By Lemma 1, $\sigma_i''' \rightsquigarrow \sigma_i'$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space. *i.e.*:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{3-i}') = \emptyset$$

Under this assumption, by Lemma 2 $\sigma_1''' \sim \sigma_2'$. Applying Lemma 2 again gives $\sigma_1' \sim \sigma_2'$.

– [APP-INFER]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ k_i = P \\ (\sigma_i \cup \{p\}), \theta, (e_a\ e_a)^p \Downarrow_{pc} \sigma_i', v \end{gathered}$$

The final antecedent of this rule joins p to the set of labels in σ_i, which means that $p \in \sigma_i'$. But by the first antecedent of this rule, $p \notin \sigma_i$. Therefore, neither P_1 nor P_2 are empty.

– [ASSIGN-NORMAL] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', a_i^{k_i} \\ k_i \neq P \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i \\ m_i = \mathit{lift}(k_i, \mathit{label}(\sigma_i'''(a_i))) \\ \sigma_i' = \sigma_i'''[a_i := v_i \sqcup m_i] \end{gathered}$$

By induction:

$$\begin{array}{ll} \sigma_1'' \sim \sigma_2'' & \sigma_1''' \sim \sigma_2''' \\ a_1^{k_1} \sim a_2^{k_2} & v_1 \sim v_2 \end{array}$$

- If $k_1 \sim k_2$ then $k_1 = k_2 = H$. By Lemma 6, $m_1 \sim m_2$. By Lemma 5, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma_1' \sim \sigma_2'$.
- Otherwise $k_1 = k_2 = L$. Then $m_1 = m_2 = L$ and hence $\sigma_1' \sim \sigma_2'$.

– [ASSIGN-UPGRADE] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \in \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', a_i^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i \\ m_i = \mathit{lift}(H, \mathit{label}(\sigma_i'''(a_i))) \\ \sigma_i' = \sigma_i'''[a_i := v_i \sqcup m_i] \end{gathered}$$

By induction:

$$\begin{array}{ll} \sigma_1'' \sim \sigma_2'' & \sigma_1''' \sim \sigma_2''' \\ a_1^{k_1} \sim a_2^{k_2} & v_1 \sim v_2 \end{array}$$

By Lemma 6 we know that $m_1 \sim m_2$. By Lemma 5, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma_1' \sim \sigma_2'$.

– [ASSIGN-INFER]: In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', a_i^{k_i} \\ k_i = P \\ (\sigma_i \cup \{p\}), \theta, (e_a := e_a)^p \Downarrow_{pc} \sigma_i', v \end{gathered}$$

The final antecedent of this rule joins p to the set of labels in σ_i, which means that $p \in \sigma_i'$. But by the first antecedent of this rule, $p \notin \sigma_i$. Therefore, neither P_1 nor P_2 are empty.

□

Restatement of Lemma 3.
Suppose $pc \neq P$ and $\sigma_1 \approx \sigma_2$ and $\theta_1 \approx \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc_i} \sigma_i', v_i$ for $i \in 1, 2$. Then $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

Proof. The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc_1} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

– [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also, $v_1 = c^{pc_1} \approx c^{pc_2} = v_2$.
– [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also $v_1 = (\theta_1(x) \sqcup pc_1) \approx (\theta_2(x) \sqcup pc_2) = v_2$.
– [FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also, $v_1 = (\lambda x.e', \theta_1)^{pc_1} \approx (\lambda x.e', \theta_2)^{pc_2} = v_2$.
– [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i', r_i^{k_i}$$

By induction, $\sigma_1' \approx \sigma_2'$ and $r_1^{k_1} \approx r_2^{k_2}$. Therefore $r_1^H \approx r_2^H$.

– [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e_a \Downarrow_{pc_i} \sigma_i'', c_i^{k_i}$$
$$\sigma_i'', \theta_i, e_a \Downarrow_{pc_i} \sigma_i', d_i^{l_i}$$
$$r_i = [\![c_i]\!](d_i)$$

By induction:

$$\sigma_1'' \approx \sigma_2'' \qquad \sigma_1' \approx \sigma_2'$$
$$c_1^{k_1} \approx c_2^{k_2} \qquad d_1^{l_1} \approx d_2^{l_2}$$

Since $c_1 = c_2$ and $d_1 = d_2$, it must be the case that $r_1 = r_2$. Therefore, $r_1^{k_1 \sqcup l_1} \approx r_2^{k_2 \sqcup l_2}$.

– [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin dom(\sigma_1) = dom(\sigma_2)$, and so $v_1 = a^{pc_1} \approx a^{pc_2} = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i'', v_i'$$
$$\sigma_i' = \sigma_i''[a := v_i']$$

By induction, $\sigma_1'' \approx \sigma_2''$ and $v_1' \approx v_2'$, and so $\sigma_1' \approx \sigma_2'$.

– [DEREF]: In this case, $e = !e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i', a_i^{k_i}$$
$$v_i = \sigma_i'(a_i) \sqcup k_i$$

By induction, $\sigma_1' \approx \sigma_2'$ and $a_1^{k_1} \approx a_2^{k_2}$. Since $\sigma_1(a_1) \approx \sigma_2(a_2)$ we know that $v_1 \approx v_2$.

– [APP-NORMAL]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$p \notin \sigma_1$$
$$\sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', (\lambda x.e_1, \theta_1')^{k_1}$$
$$k_1 \neq P$$
$$\sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1'$$
$$\sigma_1''', \theta_1'[x := v_1'], e_1 \Downarrow_{k_1} \sigma_1', v_1$$

We consider 3 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$.

- In the [APP-UPGRADE] case we have:

$$p \in \sigma_2$$
$$\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2}$$
$$\sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2'$$
$$\sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_H \sigma_2', v_2$$

By induction:

$$\sigma_1'' \approx \sigma_2''$$
$$\sigma_1''' \approx \sigma_2'''$$
$$(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$$
$$v_1' \approx v_2'$$

Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

- In the [APP-NORMAL] case, we have: where $k_2 \neq P$, then we have:

$$\begin{gathered} p \notin \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2' \\ \sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_{k_2} \sigma_2', v_2 \end{gathered}$$

By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \approx v_2' \end{gathered}$$

Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

- In the [APP-INFER] case, we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\begin{gathered} \sigma_1, \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \\ \sigma_1' \approx \sigma_2' \\ v_1 \approx v_2 \end{gathered}$$

Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

– [APP-UPGRADE]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$\begin{gathered} p \in \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', (\lambda x.e_1, \theta_1')^{k_1} \\ \sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1' \\ \sigma_1''', \theta_1'[x := v_1'], e_1 \Downarrow_H \sigma_1', v_1 \end{gathered}$$

We consider 2 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$. (The [APP-NORMAL] rule is covered above, via a symmetry argument).

- In the [APP-UPGRADE] case, we have:

$$\begin{gathered} p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2' \\ \sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_H \sigma_2', v_2 \end{gathered}$$

By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \approx v_2' \end{gathered}$$

Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

- In the [APP-INFER] case we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\begin{gathered}\sigma_1, \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \\ \sigma_1' \approx \sigma_2' \\ v_1 \approx v_2\end{gathered}$$

  Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

– [APP-INFER]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$\begin{gathered}p \notin \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', (\lambda x.e_1, \theta_1')^{k_1} \\ k = P \\ (\sigma_1 \cup \{p\}), \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma_1', v_1\end{gathered}$$

  We consider the case where evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$ is via [APP-INFER]. (The other cases are covered above, via a symmetry argument). In this case, we know that

$$\begin{gathered}p \notin \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ k_2 = P \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma_2', v_2\end{gathered}$$

  By induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

– [ASSIGN-NORMAL]: In this case, $e = (e_a := e_b)^p$, and from the antecedents of this rule, we have:

$$\begin{gathered}p \notin \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', a^{k_1} \\ k_1 \neq P \\ \sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1 \\ l_1 = \mathit{lift}(k_1, \mathit{label}(\sigma_1'''(a))) \\ \sigma_1' = \sigma_1'''[a := (v_1 \cup l_1)]\end{gathered}$$

  Without loss of generality, we assume that both evaluations allocate at the same address a. We consider 3 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$.

- In the [ASSIGN-UPGRADE] case we have:

$$\begin{gathered}p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2 \\ l_2 = \mathit{lift}(H, \mathit{label}(\sigma_2(a))) \\ \sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)]\end{gathered}$$

By induction:

$$\sigma_1'' \approx \sigma_2''$$
$$\sigma_1''' \approx \sigma_2'''$$
$$a^{k_1} \approx a^{k_2}$$
$$v_1 \approx v_2$$

Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

- In the [ASSIGN-NORMAL] case we have:

$$p \notin \sigma_2$$
$$\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2}$$
$$k_2 \neq P$$
$$\sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2$$
$$l_2 = \mathit{lift}(k_2, \mathit{label}(\sigma_2(a)))$$
$$\sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)]$$

By induction:

$$\sigma_1'' \approx \sigma_2''$$
$$\sigma_1''' \approx \sigma_2'''$$
$$a^{k_1} \approx a^{k_2}$$
$$v_1 \approx v_2$$

Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

- In the [ASSIGN-INFER] case, we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^P$. It suffices to show that

$$\sigma_1, \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1$$
$$(\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2$$
$$\sigma_1' \approx \sigma_2'$$
$$v_1 \approx v_2$$

Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

– [ASSIGN-UPGRADE]: In this case, $e = (e_a := e_b)^p$, and from the antecedents of this rule, we have:

$$p \in \sigma_1$$
$$\sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', a^{k_1}$$
$$\sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1$$
$$l_1 = \mathit{lift}(H, \mathit{label}(\sigma_1'''(a)))$$
$$\sigma_1' = \sigma_1'''[a := (v_1 \cup l_1)]$$

Without loss of generality, we assume that both evaluations allocate at the same address a. We consider 2 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$. (The [ASSIGN-NORMAL] rule is covered above, via a symmetry argument).

- In the [ASSIGN-UPGRADE] case, we have:

$$p \in \sigma_2$$
$$\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2}$$
$$\sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2$$
$$l_2 = \mathit{lift}(H, \mathit{label}(\sigma_2'''(a)))$$
$$\sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)]$$

By induction:

$$\sigma_1'' \approx \sigma_2''$$
$$\sigma_1''' \approx \sigma_2'''$$
$$a^{k_1} \approx a^{k_2}$$
$$v_1 \approx v_2$$

Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

- In the [ASSIGN-INFER] case we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\sigma_1, \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1$$
$$(\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2$$
$$\sigma_1' \approx \sigma_2'$$
$$v_1 \approx v_2$$

  Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

– [ASSIGN-INFER]: Without loss of generality, we assume that both evaluations allocate at the same address a. In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$p \notin \sigma_1$$
$$\sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', a^{k_1}$$
$$k = P$$
$$(\sigma_1 \cup \{p\}), \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1$$

We consider the case where evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2', v_2$ is via [APP-INFER]. (The other cases are covered above, via a symmetry argument). In this case, we know that

$$p \notin \sigma_2$$
$$\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2}$$
$$k_2 = P$$
$$(\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2$$

By induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

□