Program Transformation for Non-interference Verification on Programs with Pointers

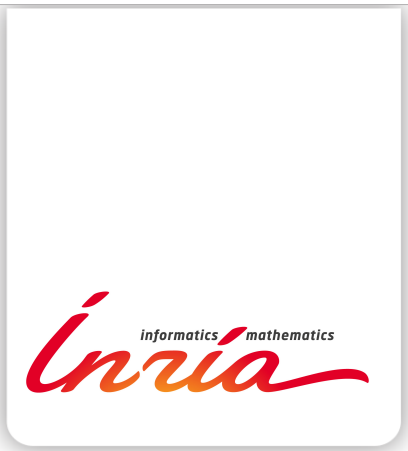
Mounir Assaf, Julien Signoles, Frédéric Tronel, Eric Totel

Program Transformation for Non-interference Verification on Programs with Pointers

Mounir Assaf, Julien Signoles, Frédéric Tronel, Éric Totel

RESEARCH REPORT
N° 8284
Avril 2013
Project-Teams CIDre

ISSN 0249-6399 ISRN INRIA/RR--8284--FR+ENG



Program Transformation for Non-interference Verification on Programs with Pointers

Mounir Assaf*, Julien Signoles*, Frédéric Tronel†, Éric Totel†

Project-Teams CIDre

Research Report n° 8284 — Avril 2013 — 33 pages

Abstract: Novel approaches for dynamic information flow monitoring are promising since they enable permissive (accepting a large subset of executions) yet sound (rejecting all unsecure executions) enforcement of non-interference. In this paper, we present a dynamic information flow monitor for a language supporting pointers. Our flow-sensitive monitor relies on prior static analysis in order to soundly enforce non-interference. We also propose a program transformation that preserves the behavior of initial programs and soundly inlines our security monitor. This program transformation enables both dynamic and static verification of non-interference.

Key-words: Non-interference, Terminating Insensitive Non-interference, Information Flow Control, Inlining, Program Transformation, Soundness, Static Analysis

* CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France, firstname.lastname@cea.fr
† Supélec, CIDre, Rennes France, firstname.lastname@supelec.fr

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Transformation de programmes pour la vérification de la non-interférence de programmes utilisant des pointeurs

Résumé : De nouvelles techniques pour le suivi dynamique de flux d'information sont prometteuses en cela qu'elles acceptent plus d'exécution que des techniques purement statiques tout en restant sûres d'utilisation (elles rejettent toute exécution qui ne vérifierait pas la propriété de non-interférence). Dans ce rapport nous présentons un moniteur dynamique de flux d'information pour un langage supportant des pointeurs. Notre moniteur est sensible aux flots de données et se base sur une analyse statique préalable afin d'assurer de manière correcte la propriété de non-interférence. Nous proposons par ailleurs une transformation de programme qui préserve la sémantique du programme initial, tout en tissant le moniteur de suivi de flux au sein du programme transformé. Ce programme transformé peut être soit exécuté, le moniteur garantissant alors le respect de la propriété de non-interférence durant l'exécution, ou bien encore être analysé statiquement à son tour pour s'assurer que toutes ses exécutions préservent cette même propriété.

Mots-clés : Non-interférence, non-interférence insensible à la terminaison, moniteur de flux d'information, transformation de programmes, analyse statique

1 Introduction

Information security is usually enforced through access control which only enforces security policies at access time. On the contrary, information flow control (IFC) ensures that propagation of information inside a program does not breach neither confidentiality nor integrity.

The seminal work in IFC is initiated by Denning and Denning [1]. They proposed a static analysis to verify that information is propagated inside programs securely with respect to a flow policy. For instance, a simple flow policy disallows leakage of secret variables into public ones, hence ensuring confidentiality. This notion is generalized by Goguen and Meseguer [2] as non-interference. Non-interference, precisely its termination-insensitive formulation (TINI), has been widely adopted in IFC as a security policy [3, 4, 5]. Informally, it states that, when changing only secret inputs, terminating executions of a program must deliver the same public outputs.

Volpano et al. [3] formalize a Denning-style static analysis as a type system for a simple imperative language. Volpano's work provides the first soundness proof stating that a typable program is secure with respect to TINI. However, Volpano's type system lacks flow-sensitivity since security labels associated to variables are not allowed to change during analysis. For example, the program $public = secret; public = 0$ is secure because the final content of variable *public* is overridden. Still, this program is not typable by Volpano's type system because of flow-insensitivity.

Hunt and Sands [6] extend Volpano's type system with flow-sensitivity, hence permitting security labels to change in order to reflect the precise security level of their contents. Introducing flow-sensitivity to security type systems contributes to more permissive security analyses. Hunt and Sands prove the soundness of their type system with respect to TINI, while typing a larger subset of secure programs in comparison with Volpano's type system.

Dynamic monitoring of information flows is also known to provide more permissiveness [7, 5] (accepting a large subset of executions). Unlike static analyses which enforce TINI for all possible execution paths, dynamic monitoring ensures that a single execution path is secure. However, permissiveness through the combination of both dynamic monitoring and flow-sensitivity requires careful examination. Indeed, Russo and Sabelfeld [5] prove that flow-sensitivity in purely dynamic IFC introduces covert channels leaking information. The main idea behind this result is that a purely dynamic monitor ignores non-executed conditional branches, missing at the same time information flows they produce. Therefore, a flow-sensitive dynamic monitor must rely on static analyses for sound (rejecting all unsecure executions) IFC.

Contributions. In this paper, we investigate **permissive yet sound flow-sensitive** IFC for programs handling pointers. Our contributions are:

- We formalize a hybrid information flow monitor for an imperative language with pointers and aliasing, by relying on a semantics built upon the Clight [8] semantics. This semantics is especially used in the CompCert [9] provably correct compiler. We prove the soundness of our monitor with respect to TINI.

- We also propose a **sound program transformation** which inlines our information flow monitor. For languages that are compiled directly into native machine code as is the case for the C language, inlining is necessary to ensure fine-grained information flow monitoring. To our knowledge, our program transformation is the first proven **sound inlining approach for dynamic monitors handling pointers**.

- Assuming the implementation of security labels and their join operator, TINI can be enforced by running the self-monitoring transformed program. This **dynamic approach**

has the advantage of being permissive since it soundly monitors a single execution path, ignoring possible unsecure paths that are not executed. The program transformation T also enables the verification of TINI by static analysis for free. Such a **static approach** computes an over-approximation of the transformed program semantics, enforcing TINI for all execution paths.

Outline. Section 2 introduces information flow background. Section 3 formalizes our information flow monitor for a simple imperative language handling pointers and aliasing. Section 4 defines a program transformation inlining our information flow monitor. We discuss related work in Section 5 and future work in Section 6.

2 Background

Non-interference. Our attacker model assumes that attackers know the source code of analyzed programs. It also supposes that attackers can only modify public inputs and read public outputs. A program is non-interferent if two terminating executions which differ only on secret inputs deliver the same public outputs. This notion of non-interference [2] formalizes independence of public outputs from secret inputs.

Information flows. Explicit and implicit flows [1] are generally taken into account when enforcing TINI. Explicit flows are produced from any source variable y assigned to a destination variable x. Implicit flows are produced whenever an affectation occurs in conditional branches. For instance, the following program $if\ (secret)\ x = 1\ else\ skip$ generates an implicit flow from *secret* to x, whatever the executed branch is. Even if x is not assigned, an attacker could learn that *secret* is false if x is different from 1. As one generally enforces a sound approximation of TINI, we suppose that assignments inside conditionals always produce implicit flows from the guards to assigned variables.

Additional information flows arise in the presence of pointers. Consider for example, the program $if\ (secret)\ \{x = \&a\}\ else\ \{x = \&b\}\ print\ *x$. An attacker, knowing the initial values of a and b, may learn information about the value of variable *secret* whenever $*x$ is output : there is an information flow from *secret* to $*x$. There are actually two different kinds of information flows involved in this case. The first one is an implicit flow from *secret* to x because of assignments inside a conditional depending on *secret*. The second one, due to pointer aliasing and dereferencing, is from x to $*x$. Thus, by transitivity, there is an information flow from *secret* to x.

Similarly, the program $if\ (secret)\ \{x = \&a\}\ else\ \{x = \&b\}\ *x = 1$ exposes pointer-induced flows from *secret* to variables a and b. An attacker having access to either variables a or b after the assignment $*x = 1$, may learn information about variable *secret*. It is worth noting that even if a (resp. b) is not assigned by instruction $*x = 1$, an information flow from *secret* to a (resp. b) is still produced. In fact, this pointer-induced information flow involves all variables that could have been written by $*x = 1$ (here, both variables a and b).

As we are aiming at enforcing TINI, we ignore in this paper all covert channels due to diverging runs and timing channels. Hence, a program like $while\ (secret)\ skip$; could leak information about variable *secret*. Yet, this is acceptable since even in the presence of outputs, Askarov et al. [4] have proved that an attacker could not know the secret in polynomial time in the size of the secret.

3 Information Flow Monitoring Semantics

Language overview. Figure 1 presents the abstract syntax of our language. It is a simple imperative language handling basic types (κ) like integers and pointers ($ptr(\tau)$). It handles aliasing but no pointer arithmetics: binary operators do not take pointers as arguments. The semantics of this language is inspired by the Clight semantics [8]. Clight is formalized in the context of the CompCert verified compiler for C programs [9].

A simplified version of the Clight big-step operational semantics considers an environment E and a memory M. $E : Var \rightharpoonup Loc$ maps variables to statically allocated locations. $M : Loc \rightharpoonup \mathbb{V}$ maps locations to values of type τ. The evaluation of an instruction c in an environment E and a memory M, denoted by $E \vdash c, M \Rightarrow M'$, results in a new memory M'. Expressions can be evaluated as either left-values or right-values depending on the position in which they occur. Only expressions having the form id or $*a$ can occur in l-value positions such as the left-hand side of assignments, whereas any expression can occur in right-value position.

As illustrated by Figure 2, l-value evaluation of expression a_1 in environment E and memory M ($E \vdash a_1, M \Leftarrow l$) provides the location l where a_1 is stored, whereas r-value evaluation of a_2 ($E \vdash a_2, M \Rightarrow v$) provides the value v of expression a_2. The assignment rule then maps the value v to the location l in the new memory M'.

Types:	$\tau ::= \kappa \mid ptr(\tau)$
Expressions:	$a ::= n \mid id \mid uop\ a \mid a_1\ bop\ a_2$
	$\mid *a \mid \&a$
Instructions:	$c ::= skip \mid a_1 = a_2 \mid c_1;\ c_2$
	$\mid if\ (a)\ c_1\ else\ c_2 \mid while\ (a)\ c$
Declarations:	$dcl ::= (\tau\ id)^*$
Programs:	$P ::= dcl;\ c$

Figure 1: Abstract syntax of our language.

$$(Assign)\ \frac{\begin{array}{c}E \vdash a_1, M \Leftarrow l \\ E \vdash a_2, M \Rightarrow v \\ M' = M[l \mapsto v]\end{array}}{E \vdash a_1 = a_2, M \Rightarrow M'}$$

Figure 2: Assignment semantics in Clight.

In order to extend Clight's three judgment rules with the information flow monitor semantics, we consider a lattice $\mathbb{S} = (SC, \sqsubseteq)$ where $public \in SC$ is the minimal element of $\mathbb{S}$. We note $\sqcup$ the associated join operator. We also consider a new kind of memory $\Gamma : Loc \rightharpoonup \mathbb{S}$, which maps locations to security labels. Informally, security memory Γ tracks the security level of locations content through tainting. For example, an assignment $x = y + z$ generates an information flow from y and z to x. Thus, Γ maps to $E(x)$ (*i.e.* the location associated to x) the security label $\Gamma(E(y)) \sqcup \Gamma(E(z))$.

Expressions. Both Clight's r-value and l-value evaluations of expressions are extended to support the propagation of security labels, as illustrated in Figure 3: the evaluation of expressions yields both a value $v \in \mathbb{V}$ and a security label $s \in S$. If the pair (l, s_l) is the result of l-value evaluation of expression a, then the security label s_l captures pointer-induced flows produced by possible dereferences occurring in a, whereas $s_r = \Gamma(l)$ captures explicit flows produced by reading the value $M(l)$ of a. Therefore, the r-value evaluation of a produces a value $v = M(l)$ and a security label $s = s_l \sqcup s_r$ taking into account both explicit and pointer-induced flows through the join operator (rule RV). Note that the semantics of Clight expressions can be obtained from Figure 3 by ignoring all the monitor related operations.

The security label associated to the r-value of a defines the label associated to the l-value of $*a$ (rule LV_{MEM}), hence taking into account the pointer-induced information flow from a to $*a$. R-values of constants are labeled as *public* because attackers are supposed to know the source

$$LV_{ID}\ \frac{E(id) = l}{E \vdash id, M, \Gamma \Leftarrow l, public} \qquad LV_{MEM}\ \frac{E \vdash a, M, \Gamma \Rightarrow ptr(l), s}{E \vdash *a, M, \Gamma \Leftarrow l, s}$$

$$RV_{CONST}\ E \vdash n, M, \Gamma \Rightarrow n, public \qquad RV\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s_l \quad M(l) = v \quad s_r = \Gamma(l) \quad s = s_l \sqcup s_r}{E \vdash a, M, \Gamma \Rightarrow v, s}$$

$$RV_{REF}\ \frac{E \vdash a, M, \Gamma \Leftarrow l, s}{E \vdash \&a, M, \Gamma \Rightarrow ptr(l), s} \qquad RV_{UOP}\ \frac{E \vdash a, M, \Gamma \Rightarrow v, s \quad uop\ v = v'}{E \vdash uop\ a, M, \Gamma \Rightarrow v', s}$$

$$RV_{BOP}\ \frac{E \vdash a_1, M, \Gamma \Rightarrow v_1, s_1 \quad E \vdash a_2, M, \Gamma \Rightarrow v_2, s_2 \quad v_1\ bop\ v_2 = v \quad s_1 \sqcup s_2 = s}{E \vdash a_1\ bop\ a_2, M, \Gamma \Rightarrow v, s}$$

Figure 3: Information flow monitor semantics of expressions.

code. Since the locations of variables are at known offsets from the base pointer, we associate *public* to the l-values of variables (rule LV_{ID}). The label of the l-value of a defines the label associated to the r-value of $\&a$ (rule RV_{REF}). The security label associated to the r-value of a is propagated to the r-value of $uop\ a$ (rule RV_{UOP}). Likewise, the security label associated to the r-value of $a_1\ bop\ a_2$ takes into account both a_1 and a_2 r-values security labels through the join operator.

Figure 4 illustrates an example of the r-value evaluation of $*x$. Supposing that x is stored at location l_x and points to a variable a stored at location l_a, the r-value evaluation of $*x$ takes into account both pointer-induced and explicit flows since both s_x (the security label of x) and s_a (the security label of a) affect the resulting security label s.

$$RV\ \frac{LV_{MEM}\ \dfrac{RV\ \dfrac{LV_{ID}\ \dfrac{E(x) = l_x}{E \vdash x, M, \Gamma \Leftarrow l_x, public} \quad M(l_x) = ptr(l_a) \quad \Gamma(l_x) = s_x \quad s_x = public \sqcup s_x}{E \vdash x, M, \Gamma \Rightarrow ptr(l_a), s_x}}{E \vdash *x, M, \Gamma \Leftarrow l_a, s_x} \quad M(l_a) = v \quad \Gamma(l_a) = s_a \quad s = s_a \sqcup s_x}{E \vdash *x, M, \Gamma \Rightarrow v, s}$$

Figure 4: An example of expression $*x$ evaluation.

One consequence of rules LV_{ID} and RV_{REF} is that addresses of variables are labeled as *public*. Thus, they can be accessed by attackers and used to bypass security measures such as ASLR (Address Space Layout Randomization). In fact, this kind of information leaks is out of scope for our analysis since addresses of variables are not secret inputs of programs. Furthermore, mapping any security label s other than *public* to the l-value of variables id would taint all data accessed through dereferences of id, causing a label creep problem [10].

Instructions. The semantics of instructions is presented in Figure 5. It is a combination of dynamic monitoring and static analysis through the use of $S_P(c)$, the set of locations that may have been written by instruction c of program P. The statically computed set $S_P(c)$ is fed to the semantics whenever a call to the *update* operator occurs. We also introduce a new meta-variable

$$(Assign)\ \frac{\begin{array}{c}E \vdash a_1, M, \Gamma \Leftarrow l_1, s_1 \quad E \vdash a_2, M, \Gamma \Rightarrow v_2, s_2 \\ s = s_1 \sqcup s_2 \sqcup \underline{pc} \quad s' = s_1 \sqcup \underline{pc} \quad M' = M[l_1 \mapsto v_2] \\ \Gamma'' = \Gamma[l_1 \mapsto s] \quad \Gamma' = update(a_1 = a_2, s', \Gamma'')\end{array}}{E \vdash a_1 = a_2, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'} \qquad (Comp)\ \frac{\begin{array}{c}E \vdash c_1, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1 \\ E \vdash c_2, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2\end{array}}{E \vdash c_1; c_2, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2}$$

$$(If_{tt})\ \frac{\begin{array}{c}E \vdash a, M, \Gamma \Rightarrow v, s \quad istrue(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad E \vdash c_1, M, \Gamma, \underline{pc'} \Rightarrow M_1, \Gamma_1 \\ \Gamma'_1 = update(c_2, \underline{pc'}, \Gamma_1)\end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma'_1} \qquad (W_{ff})\ \frac{\begin{array}{c}E \vdash a, M, \Gamma \Rightarrow v, s \quad isfalse(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad \Gamma' = update(c, \underline{pc'}, \Gamma)\end{array}}{E \vdash while\ (a)\ c, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma'}$$

$$(If_{ff})\ \frac{\begin{array}{c}E \vdash a, M, \Gamma \Rightarrow v, s \quad isfalse(v) \\ \underline{pc'} = s \sqcup \underline{pc} \quad E \vdash c_2, M, \Gamma, \underline{pc'} \Rightarrow M_2, \Gamma_2 \\ \Gamma'_2 = update(c_1, \underline{pc'}, \Gamma_2)\end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma'_2} \qquad (W_{tt})\ \frac{\begin{array}{c}E \vdash a, M, \Gamma \Rightarrow v, s \quad istrue(v) \\ \underline{pc'} = s \sqcup \underline{pc} \\ E \vdash c, M, \Gamma, \underline{pc'} \Rightarrow M', \Gamma'\end{array}}{E \vdash while\ (a)\ c, M', \Gamma', \underline{pc} \Rightarrow M'', \Gamma''}$$

$$(Skip)\ E \vdash skip, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma \qquad update(c, s, \Gamma) \triangleq \begin{cases}\Gamma(l) & \forall l \notin S_P(c) \\ \Gamma(l) \sqcup s & \forall l \in S_P(c)\end{cases}$$

Figure 5: Information flow monitor big-step semantics of instructions.

$$(Assign)\ \frac{\begin{array}{c}E \vdash *x, M, \Gamma \Leftarrow l_a, s_x \quad E \vdash 1, M, \Gamma \Rightarrow 1, public \quad s = s_x \sqcup public \sqcup \underline{pc} \\ s' = s_x \sqcup \underline{pc} \quad M' = M[l_a \mapsto 1] \quad \Gamma'' = \Gamma[l_a \mapsto s] \quad \Gamma' = update(*x = 1, s', \Gamma'')\end{array}}{E \vdash *x = 1, M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'}$$

Figure 6: An example of instruction $*x = 1$ evaluation.

$\underline{pc}$ to capture implicit flows. $\underline{pc}$ can be viewed as the security label of the program counter. Each time a program enters a conditional, $\underline{pc}$ is updated with the guard security label in order to reflect generated implicit flows. Therefore, evaluation of instructions occurs in a memory Γ, an execution context $\underline{pc}$ in addition to a memory M and an environment E. It produces new memories Γ' and M'.

For assignment $a_1 = a_2$ (rule *Assign*), the join of three security labels are mapped to the location of a_1. First, s_1 takes into account pointer-induced flows from the l-value of a_1. Second, s_2 considers explicit flows from the r-value of a_2. Third, $\underline{pc}$ captures the implicit flows generated by conditionals. Additionally, assignments generate pointer-induced flows from the l-value of a_1 to the set of possibly written locations. Consequently, the *update* operator propagates the union of $\underline{pc}$ and s_1 to $S_p(a_1 = a_2)$. Assuming that x points to a variable a stored at location l_a, Figure 6 illustrates the evaluation of instruction $*x = 1$. The security label s_x (resp. pc) affects the security label of variable a in order to take into account pointer-induced flows (resp. implicit flows). Finally, the *update* operator propagates the security label s' to the set $S_P(*x = 1)$ to capture pointer-induced flows due to the assignment $*x = 1$.

For conditionals (rules If_{tt} and If_{ff}), a new context of execution $\underline{pc'}$ takes into account implicit flows generated by the conditional guard a. When a is evaluated to *true* (rule If_{tt}, the other one is symmetrical), the resulting security environment takes into account the implicit flows induced by both the executed branch c_1 and the non-executed one c_2. Implicit flows in c_1

are computed by the evaluation of c_1 in $\underline{pc'}$, whereas the *update* operator handles the ones from c_2 by propagating $\underline{pc'}$ to the set $S_P(c_2)$. Rules W_{tt} and W_{ff} are similar to conditional rules. Finally, a sequence of instructions $c_1; c_2$ is executed in the same execution context (rule $Comp$).

Soundness. In order to formalize TINI, Definition 1 introduces an equivalence relation for memories: two memories M_1 and M_2 are s-equivalent if they are equal for the set of locations l whose label $\Gamma(l)$ is at most s.

Definition 1 (Equivalence relation $\sim^s_\Gamma$) *For all Γ, $s \in \mathbb{S}$, M_1, M_2, M_1 and M_2 are s-equivalent ($M_1 \sim^s_\Gamma M_2$) if and only if*

$$\forall l \in Loc, \Gamma(l) \sqsubseteq s \implies M_1(l) = M_2(l).$$

Non-interference, by Definition 2, ensures that an attacker knowing only inputs and outputs up to a security level s cannot gain any knowledge of inputs whose security levels are higher than s.

Definition 2 (Termination-insensitive non-interference)
For all $c, E, \Gamma, M_1, M'_1, M_2, M'_2, s, \underline{pc} \in \mathbb{S}$, such that
$E \vdash c, M_1, \Gamma, \underline{pc} \Rightarrow M'_1, \Gamma'_1$ and $E \vdash c, M_2, \Gamma, \underline{pc} \Rightarrow M'_2, \Gamma'_2$,

$$M_1 \sim^s_\Gamma M_2 \implies \Gamma'_2 = \Gamma'_1 = \Gamma' \text{ and } M'_1 \sim^s_{\Gamma'} M'_2.$$

This definition of non-interference is termination-insensitive since it ignores behaviors of diverging runs, including information leaks due to the attacker ability to observe (non-)termination of programs. Definition 2 is equivalent to the definitions of TINI in the literature [3, 4, 6, 11]. Moreover, our definition of non-interference is equivalent to what Askarov et al. [4] call batch job TINI, since attackers are not allowed to know intermediate results of computation through outputs.

Theorem 1 (Soundness) *The information flow-extended semantics is sound with respect to termination-insensitive non-interference as defined in 2.*

Theorem 1 proves that our monitor semantics is sound with respect to TINI [1]. The proof, by induction on instructions evaluation $\Rightarrow$, relies on the fact that both l-value and r-value evaluations of expressions in s-equivalent memories yield the same result for expressions whose label is below s. This theorem also proves that attackers cannot learn information by observing the behavior of our monitor since it ensures that both output security memories are equal.

4 Program Transformation

This section presents an inlining approach for our monitoring semantics as a program transformation. This approach has the benefits of enabling both static and dynamic analysis since both analyses can be considered depending on the required level of confidence. The former would focus on soundness by ensuring that all execution paths of the analysed program are secure. The latter would emphasize on permissiveness by enforcing non-interference for the execution path of a single run.

[1] Full details of our proofs can be found in the appendix.

Informally, the program transformation maps a shadow variable —a security label— to each variable of $Var(P)$, the set of variables of the initial program P. Inlining our monitor then consists of propagating those security labels with respect to the monitor semantics. For this reason, types of our language are extended with a type τ_s representing security labels. Expressions are extended with security labels denoted s and a join operator $\sqcup$ on security labels. The range of memories M is also extended to $\mathbb{V} \cup \mathbb{S}$.

In order to handle pointers, we introduce in Definitions 3 and 4 the depth $\mathscr{D}(id)$ of a variable id and a bijection $\Lambda(id, k)$, with $k \in [0, \mathscr{D}(id)]$. $\mathscr{D}(id)$ is the number of dereferences such that $*^{\mathscr{D}(id)}id$ yields a basic type κ, whereas Λ maps each initial variable id to $\mathscr{D}(id)$ different shadow variables. Basically, $*^k\Lambda(id, k)$ is the security label of $*^k id$.

Definition 3 (Depth $\mathscr{D}(x)$ of variable x)

Let τ_x be the type of variable $x \in Var(P)$. $\mathscr{D}(x) = \mathscr{D}(\tau) = \begin{cases} 0 & \text{if } \tau = \kappa \\ 1 + \mathscr{D}(\tau') & \text{if } \tau = ptr(\tau') \end{cases}$

Definition 4 (Bijection Λ)

$\Lambda : \{(x, k) : x \in Var(P) \text{ and } k \in [0, \mathscr{D}(x)]\} \to Var'$ *such that* $Var' \subset Var \setminus Var(P)$ *is a bijection mapping to each initial variable x exactly $\mathscr{D}(x)$ shadow variables, denoted $\Lambda(x, k)$, such that $\Lambda(x, k)$ has a type $ptr^{(k)}(\tau_s)$.*

We extend Λ to all l-value expressions ($\Lambda(*^r x, k) \triangleq *^r\Lambda(x, k + r)$) such that $\Lambda(*^k id, 0)$ is equal to $*^k\Lambda(id, k)$. Hence $\Lambda(*^k id, 0)$ also captures the security label of $*^k id$.

Our program transformation, denoted T, maintains a pointer-related invariant in order to correctly handle aliasing. Essentially, if x points to an integer variable a, shadow variable $\Lambda(x, 1)$ also points to $\Lambda(a, 0)$. This way, whenever we read (or write) the same integer through $*x$ or a, we also read (or write) the same security label through either $*\Lambda(x, 1)$ or $\Lambda(a, 0)$. Listings 1 and 2 illustrate an example of our program transformation. Instruction 3 in Listing 1 is transformed into instructions 3, 4, 5 and 6 in Listing 2. Instructions 3, 5 and 6 of the transformed program reproduce the semantics of *Assign* rules as defined in the monitoring semantics (Figure 5), whereas instruction 4 maintains the aliasing invariant. Thanks to instructions 4 and 9 of the transformed program, instruction 13 updates the correct security label during execution.

Listing 1: The initial program.

```
1 //S_P(c3)=S_P(c5)={E(x)}
2 if (secret)
3   x = &a;
4 else
5   x = &b;
6 //S_P(c7)={E(a),E(b)}
7 *x = 1
```

Listing 2: The transformed program.

```
1  pc'=pc ⊔ Λ(secret,0);
2  if (secret) {
3    Λ(x,0) = public;
4    Λ(x,1) = &Λ(a,0);
5    Λ(x,0) = Λ(x,0) ⊔ pc';
6    x = &a;
7  } else {
8    Λ(x,0) = public;
9    Λ(x,1) = &Λ(b,0);
10   Λ(x,0) = Λ(x,0) ⊔ pc';
11   x = &b;
12 }
13 *Λ(x,1) = Λ(x,0) ⊔ public;
14 Λ(a,0) = Λ(x,0) ⊔ pc;
15 Λ(b,0) = Λ(x,0) ⊔ pc;
16 *x = 1
```

As in Definition 5, two expressions are aliased in memory M if their l-value evaluation yields the same location. Hence, the aliasing invariant, stated as Lemma 1, ensures that two l-value expressions are aliased iff their shadow variables are aliased.

Definition 5 (Aliasing equivalence relation $\sim^M_{lval}$)

For all $a_1, a_2 \in Exp$, for all E, M such that $E \vdash a_1, M \Leftarrow l_1$ and $E \vdash a_2, M \Leftarrow l_2$.

$$a_1 \sim^M_{lval} a_2 \iff l_1 = l_2$$

Lemma 1 (Aliasing invariant)
For all $E, c, M, M', \Gamma, \Gamma', \underline{pc}, pc$ *such that* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
Let the predicate $\Omega(M) \triangleq \forall x, y \in Var(P)$, *for all* $r \in [0, \mathscr{D}(y)]$,

$$x \sim^{M}_{lval} *^{r} y$$

$$\iff \forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^{M}_{lval} \Lambda(*^{r} y, k)$$

Then $\Omega(M) \implies \Omega(M')$.

Transformation T relies on Definition 6 of operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$ which express security labels of expressions in terms of shadow variables. They respectively capture the label of the l-value of a, the label of the r-value of a, and $\Gamma(l_a)$, where l_a is the location of a. They accurately reproduce the monitoring semantics for expressions as defined in Figure 3.

Definition 6 (Operators $\mathcal{L}_L$, $\mathcal{L}_R$ and $\mathcal{L}$)

$$\mathcal{L}_R(n) \triangleq public \qquad \mathcal{L}_R(uop\ a) \triangleq \mathcal{L}_R(a) \qquad \mathcal{L}_R(\&a) \triangleq \mathcal{L}_L(a) \qquad \mathcal{L}_R(a_1\ bop\ a_2) \triangleq \mathcal{L}_R(a_1) \sqcup \mathcal{L}_R(a_2)$$

$$\mathcal{L}_R(a) \triangleq \mathcal{L}_L(a) \sqcup \mathcal{L}(a) \qquad \mathcal{L}(a) \triangleq \Lambda(a, 0) \qquad \mathcal{L}_L(id) \triangleq public \qquad \mathcal{L}_L(*a) \triangleq \mathcal{L}_R(a)$$

The l-values of a variable id is associated with the security label *public* (rule LV_{ID}), so does $\mathcal{L}_L(id)$. $\mathcal{L}_L(*a)$, the security label associated to the l-value $*a$, is defined as $\mathcal{L}_R(a)$, the security label associated to the r-value of a (rule LV_{MEM}). As for r-values (rule RV_{CONST}), the security label of constant integers $\mathcal{L}_R(n)$ is defined as *public*. The security label of r-values expressions $\mathcal{L}_R(a)$ is defined as the join of their l-value label $\mathcal{L}_L(a)$ and the label of their content $\mathcal{L}(a)$ (rule RV) in order to take into account both pointer-induced and explicit flows. $\mathcal{L}_R(\&a)$, the label of r-value expressions $\&a$ is defined as $\mathcal{L}_L(a)$, the label of the l-value a (rule RV_{REF}). $\mathcal{L}_R(uop\ a)$ and $\mathcal{L}_R(a_1\ bop\ a_2)$ are respectively defined according to rules RV_{UOP} and RV_{BOP}. Finally, the label $\mathcal{L}(a)$ associated to the content of a is defined as $\Lambda(a, 0)$, which represents $\Gamma(l_a)$ in the monitoring semantics. Figure 7 illustrates the computation of the label associated to a r-value $*x$. Intuitively, for the transformation to be correct, we must ensure that the evaluation of $\Lambda(x, 0)$ and $*\Lambda(x, 1)$ in M respectively results in $s_x = \Gamma(l_x)$ and $s_a = \Gamma(l_a)$.

$$\begin{aligned}\mathcal{L}_R(*x) &= \mathcal{L}_L(*x) \sqcup \mathcal{L}(*x) = \mathcal{L}_R(x) \sqcup \Lambda(*x, 0) = \mathcal{L}_L(x) \sqcup \mathcal{L}(x) \sqcup *\Lambda(x, 1) \\ &= public \sqcup \Lambda(x, 0) \sqcup *\Lambda(x, 1)\end{aligned}$$

Figure 7: An example of security label computation by both semantics and transformation T.

We present the program transformation rules in Figure 8. For brevity, $c_k; \forall k \in [0, n]$ denotes the sequence of instructions $c_0; c_1; \ldots c_n$. Since the transformation T must maintain the execution context and must propagate it to all possibly written locations in non-executed branches, it creates for each conditional and loop a new shadow variable of type τ_s, denoted pc'. Variable pc' captures the new execution context $\underline{pc'}$ defined in the semantics. The transformation then parameterizes the branches with the new shadow variable pc'. It also uses the inverse of environment E, denoted E^{-1}, in order to find the set of variables corresponding to the locations $l \in S_P(c)$. Then it propagates the execution context pc' to all the corresponding shadow variables. This way, the program transformation reproduces the semantics of the *update* operator for conditionals and loops. Note that E^{-1} is well defined since each location has only one corresponding declared variable. We are confident that even for further extensions including dynamically allocated locations, we should be able to find a corresponding shadow expression if there is an expression pointing to that location.

$$T[skip, pc] \mapsto skip \qquad T[c_1;\ c_2, pc] \mapsto T[c_1, pc];\ T[c_2, pc]$$

$$T[a_1 = a_2, pc] \mapsto \begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 = a_2) \\ a_1 = a_2; \end{cases}$$

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto \begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

$$T[while\ (a)\ c, pc] \mapsto \begin{cases} while\ (a)\ \{ \\ \quad pc' = \mathcal{L}_R(a) \sqcup pc; \\ \quad T[c, pc']; \\ \} \\ pc' = \mathcal{L}_R(a) \sqcup pc; \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c) \end{cases}$$

Figure 8: Program transformation semantics.

For assignments $a_1 = a_2$, the program transformation propagates three security labels to the shadow expression of a_1 according the monitor semantics. Since assignments create new aliasing relations, transformation T also generates $\mathscr{D}(a_1)$ assignments to maintain the aliasing invariant stated in Lemma 1. Finally, T uses E^{-1} and Λ to find shadow variables corresponding to locations in $S_P(c)$ and taints them with the security label $\mathcal{L}_L(a_1) \sqcup pc$.

The transformed program $T(P)$ is behaviourally equivalent to the initial program P. Let $E|_{var(P)}$ (resp. $M|_{Loc(P)}$ and $\Gamma|_{Loc(P)}$) be the restriction of environment E (resp. of memory M and Γ) to the set $Var(P)$ of initial variables (resp. to the set $Loc(P)$ of initial locations). More precisely, Theorem 2 states that for any terminating run, executions of P and $T(P)$ in equal input memories for initial locations $Loc(P)$ result in equal memories for those same locations. The proof by induction on instructions evaluation relies on the fact that program transformation T introduces only assignments handling shadow variables. Hence, those additional assignments do not modify neither values nor security labels associated to the set $Loc(P)$ of initial locations.

Theorem 2 (Initial semantics preservation) *For all $c, E, M, \Gamma, \underline{pc}, pc$ such that:*
$E|_{Var(P)} \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1, \Gamma_1$ *and* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$.

$$\textit{Then, } M_2|_{Loc(P)} = M_1 \textit{ and } \Gamma_2|_{Loc(P)} = \Gamma_1.$$

Theorem 3 proves the soundness of the transformation T with respect to the monitor semantics presented in Figure 5. Informally, the theorem supposes that values of shadow variables

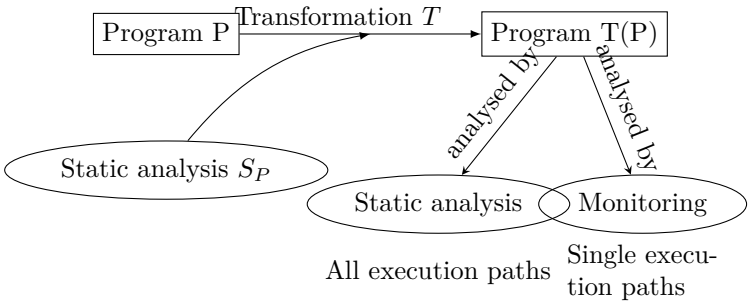


Figure 9: Non-interference verification using the program transformation T.

(resp. execution context variable pc) are initialized according to the initial security memory Γ (resp. execution context $\underline{pc}$). Then after the execution of the transformed instructions, it states that the values of shadow variables capture the exact values of the output security memory.

Theorem 3 (Sound monitoring of information flows) *Let c, for all $E, M, \Gamma, M', \Gamma'$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.*

Let us define the predicate $\Upsilon(E, M, \Gamma) \triangleq$ for all $x \in Var(P)$, for all $k \in [0, \mathscr{D}(x)]$, $E \vdash *^k x, M \Leftarrow l_{xk}$ and $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}$.

The following result holds: $\Upsilon(E, M, \Gamma)$ and $E \vdash pc, M \Rightarrow \underline{pc} \implies \Upsilon(E, M', \Gamma')$.

As the program transformation is sound with respect to our information flow monitor semantics, it is also sound wrt. TINI. Therefore, we can soundly reason about information flows through security labels defined by this program transformation. To our knowledge, that is the first proof of soundness for inlining information flow monitors handling pointers with aliasing. The proof, by induction on instructions evaluation $\Rightarrow$, heavily relies on the aliasing invariant stated in Lemma 1.

TINI verification. Figure 9 shows that the program transformation T can be used to verify TINI through both dynamic and static analysis. Assuming the implementation of security labels and their join operator, running the self-monitoring program $T(P)$ enforces TINI dynamically —actually, this is a hybrid approach since the monitor relies on a prior static analysis S_P— for single execution paths. This dynamic approach has the advantage of being permissive since it ignores possible unsecure paths that are not executed. It also enables dynamic loading of security policies [12], taking into account eventual updates. The transformation T also enables the verification of TINI by static analysis: for instance, off-the-shelf abstract interpretation tools can compute an over-approximation of $T(P)$ semantics for all execution paths, without implementing new abstract domains. While still being more permissive than traditional type systems, such an approach freezes the enforced security policy. Yet, it enhances our confidence in the analyzed program. It also completely lifts the burden of runtime overhead.

5 Related Work

Information flow monitors. Le Guernic et al. [7] formalize a sound flow-sensitive monitor for a simple imperative language with outputs. Le Guernic's monitor combines both static and dynamic analysis in order to enforce TINI. It is based on edit automata [13], which are monitors enforcing a security policy by modifying program actions, namely changing secret outputs to

default values in Le Guernic's monitor. Extending our approach with outputs is straightforward. Le Guernic et al. suggest that their monitor can be implemented as a program transformation or a virtual machine (VM).

Russo and Sabelfeld [5] parameterize their hybrid monitor for a simple imperative language by different enforcement actions (default, failstop or suppress). They also prove the necessity to rely on static analysis to soundly monitor information flows while still being more permissive than Hunt-Sands-style [3] flow-sensitive type systems. Unlike monitors based on Russo and Sabelfeld's one, we use a big-step semantics. Hence, we neither need to maintain a stack of security labels for execution contexts, nor insert instructions to notify the monitor at the immediate postdominator of each conditional.

Moore and Chong [14] extend the VM-like monitor of Russo and Sabelfeld with dynamically allocated references, allowing different sound memory abstractions. In our semantics, we use the most precise instantiation of their memory abstraction where each concrete location correspond to one abstract location. While it is undecidable in the general case to determine which locations might be updated by an instruction, we argue that, for the sake of permissiveness, it is necessary to be as precise as possible at least for the set of finite statically allocated locations.

Austin and Flanagan [11] investigate a purely dynamic monitor for a λ-calculus language with references. Their monitor supports a limited flow-sensitivity since it implements a conservative no-sensitive upgrade policy; the monitor stops the execution when assigning a public variable in a secret context. Thus, their monitor is proven sound without having to rely on static analyses. Austin and Flanagan [15] also enhance their monitor by a permissive-upgrade approach; their monitor labels public data that is assigned in secret contexts as partially leaked, then soundly forbid branching on those data. Our monitor is fully flow-sensitive, hence more permissive.

Sound inlining. Chudnov and Naumann [16] design a sound monitor inlining approach based on Russo and Sabelfeld's monitor. As they aim at monitoring information flows for Javascript, they argue that VM monitors are impractical because of just-in-time compilation. Their language supports output instructions but no references. We also believe that inlining is necessary when the language is compiled rather than interpreted.

Magazinius et al. [17] investigate sound inlining of security monitors for an imperative language supporting dynamic code evaluation but no references. Their monitor is purely dynamic since it uses a no-sensitive upgrade policy as in Austin and Flanagan [11]. Our program transformation approach can also be applied for such a policy in order to soundly monitor information flows for richer languages, including pointers.

6 Conclusion and future work

We have formalized a sound flow-sensitive information flow monitor handling pointers and aliasing. We have also inlined our monitor through a program transformation proven sound with respect to our monitor semantics, hence with TINI. Our program transformation enables permissive yet sound enforcement of TINI by both dynamic and static analyses. Our monitor semantics ignores diverging runs since it is inspired by a simple version of the Clight big-step semantics stripped of coinduction [8]. As pointed by Le Guernic [7], this is not problematic when dealing with TINI because we ignore non-termination covert channels.

As we aim to support a large subset of the C language, we plan on extending both the semantics and the program transformation with richer C constructs. We are currently implementing our program transformation as a Frama-C plug-in, an open-source tool for modular analysis of C programs [18]. Frama-C enables the design of powerful analyses relying on the collaboration

of off-the-shelf plug-ins. We are going to rely on Value Analysis [19], an abstract interpretation plug-in of Frama-C, in order to compute a correct approximation $S_P(c)$, of the set of locations that might be updated by an instruction c. Frama-C also supports ACSL [20], a formal specification language for C programs. This language can allow us to handle declassification annotations.

References

[1] Denning, D., Denning, P.: Certification of Programs for Secure Information Flow. Communications of the ACM **20**(7) (1977) 504–513

[2] Goguen, J., Meseguer, J.: Security Policies and Security Models. In: IEEE Symposium on Research in Security and Privacy. (1982)

[3] Volpano, D., Irvine, C., Smith, G.: A Sound Type System for Secure Flow Analysis. Journal in Computer Security **4**(2-3) (1996) 167–187

[4] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-Insensitive Noninterference Leaks More Than Just a Bit. In: Computer Security - ESORICS 2008. Volume 5283 of Lecture Notes in Computer Science. (2008)

[5] Russo, A., Sabelfeld, A.: Dynamic vs. Static Flow-Sensitive Security Analysis. Computer Security Foundations Symposium, IEEE **0** (2010) 186–199

[6] Hunt, S., Sands, D.: On Flow-Sensitive Security Types. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Volume 41., ACM (2006) 79–90

[7] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-Based Confidentiality Monitoring. Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues (2007) 75–89

[8] Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. Journal of Automated Reasoning **43**(3) (2009) 263–288

[9] Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM **52**(7) (2009) 107–115

[10] Sabelfeld, A., Myers, A.: Language-Based Information-Flow Security. Selected Areas in Communications, IEEE Journal on **21**(1) (2003) 5–19

[11] Austin, T., Flanagan, C.: Efficient Purely-Dynamic Information Flow Analysis. ACM Sigplan Notices **44**(8) (2009) 20–31

[12] Chandra, D., Franz, M.: Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, IEEE (2007) 463–475

[13] Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security **4**(1) (2005) 2–16

[14] Moore, S., Chong, S.: Static Analysis for Efficient Hybrid Information-Flow Control. In: Computer Security Foundations Symposium (CSF), 2011 IEEE 24th, IEEE (2011) 146–160

[15] Austin, T.H., Flanagan, C.: Permissive Dynamic Information Flow Analysis. In: PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, ACM (2010) 1–12

[16] Chudnov, A., Naumann, D.: Information Flow Monitor Inlining. In: Computer Security Foundations Symposium (CSF), 2010 23rd IEEE, IEEE (2010) 200–214

[17] Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly Inlining of Dynamic Security Monitors. Computers & Security (2011)

[18] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Program Analysis Perspective. Software Engineering and Formal Methods (2012) 233–247

[19] Cuoq, P., Prevosto, V., Yakobowski, B.: Frama-C's Value Analysis Plug-in. (September 2012) `http://frama-c.com/download/frama-c-value-analysis.pdf`.

[20] Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (September 2012) URL: http://frama-c.cea.fr/acsl.html.

Appendix

We prove that our monitor is sound with respect to TINI. We take advantage of the symmetry of both runs by introducing a partial order relation 7 on security memories Γ. Definition 8 is equivalent to Definition 2 when both input security memories and both execution contexts are equal in both runs.

Definition 7 (Less restrictive up to label s ($\sqsubseteq_s$))
For all $s \in \mathbb{S}$, all Γ_1, Γ_2, Γ_2 is less restrictive than Γ_1 up to security label s ($\Gamma_2 \sqsubseteq_s \Gamma_1$) iff for all $l \in Loc$,

$$\Gamma_1(l) \sqsubseteq s \text{ implies } \Gamma_2(l) \sqsubseteq \Gamma_1(l)$$

Definition 8 (Termination-insensitive non-interference)
For all $c, E, \Gamma_1, \Gamma_2, M_1, M_1', M_2, M_2'$, for all $s, \underline{pc}_1, \underline{pc}_2 \in \mathbb{S}$, such that $E \vdash c, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M_1', \Gamma_1'$ and $E \vdash c, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M_2', \Gamma_2'$, then

$$\underline{pc}_2 \sqsubseteq \underline{pc}_1 \text{ and } \Gamma_2 \sqsubseteq_s \Gamma_1 \text{ and } M_1 \sim^s_{\Gamma_1} M_2 \text{ implies } \Gamma_2' \sqsubseteq_s \Gamma_1' \text{ and } M_1' \sim^s_{\Gamma_1'} M_2'$$

We introduce Lemma 2, which proves that for two s-equivalent memories, if the l-value evaluation of an expression yields a label below s, then it is evaluated to the same location in the second run, and to a label that is less restrictive than the label of the first run.

Lemma 2 (L-value evaluation in s-equivalent memories)
For all $E, M_1, M_2, \Gamma_1, \Gamma_2, s \in \mathbb{S}$, such that $\Gamma_2 \sqsubseteq_s \Gamma_1$ and $M_1 \sim^s_{\Gamma_1} M_2$,
for all $a \in Exp$ such that $E \vdash a, M_1, \Gamma_1 \Leftarrow l_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Leftarrow l_2, s_2$
then
$s_1 \sqsubseteq s$ *implies that* $l_1 = l_2$ *and* $s_2 \sqsubseteq s_1$

Proof:
Let: $E, M_1, M_2, \Gamma_1, \Gamma_2$ and $s \in \mathbb{S}$.
Let: $a \in Exp$ such that $E \vdash a, M_1, \Gamma_1 \Leftarrow l_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Leftarrow l_2, s2$.
Assume: 1. $\Gamma_2 \sqsubseteq_s \Gamma_1$
2. $M_1 \sim^s_{\Gamma_1} M_2$
3. $s_1 \sqsubseteq s$
Prove: 1. $l_1 = l_2$
2. $s_2 \sqsubseteq s_1$
Proof sketch: By induction on l-value evaluations of expressions.
$\langle 1 \rangle 1$. Case: LV_{id}
$\langle 2 \rangle 1$. $s_2 = s_1 = public$
$\langle 3 \rangle 1$. Q.E.D.
Proof: By rule LV_{id}, labels associated to l-values of variables are defined as the bottom of $\mathbb{S}$
$\langle 2 \rangle 2$. $E(a) = l_1 = l_2$
$\langle 3 \rangle 1$. Q.E.D.
Proof: By rule LV_{id}. Environment E is the same for both runs.
$\langle 2 \rangle 3$. Q.E.D.
Proof: By $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$.
$\langle 1 \rangle 2$. Case: LV_{MEM}
Let: l_a, s_l, l, s_r and l_a', s_l', l', s_r' such that

$$LV_{MEM}\ \frac{RV\ \dfrac{(\mathbf{1})E \vdash a, M_1, \Gamma_1 \Leftarrow l_a, s_l \qquad M_1(l_a) = ptr(l_1) \quad \Gamma_1(l_a) = s_r \qquad s_1 = s_l \sqcup s_r}{E \vdash a, M_1, \Gamma_1 \Rightarrow ptr(l_1), s_1}}{E \vdash *a, M_1, \Gamma_1 \Leftarrow l_1, s_1}$$

$$LV_{MEM}\ \frac{RV\ \dfrac{E \vdash a, M_2, \Gamma_2 \Leftarrow l'_a, s'_l \qquad M_2(l'_a) = ptr(l_2) \quad \Gamma_2(l'_a) = s'_r \qquad s_2 = s'_l \sqcup s'_r}{E \vdash a, M_2, \Gamma_2 \Rightarrow ptr(l_2), s_2}}{E \vdash *a, M_2, \Gamma_2 \Leftarrow l_2, s_2}$$

SUFFICES ASSUME: $s_1 \sqsubseteq s$
PROVE: 1. $s_2 \sqsubseteq s_1$
2. $l_2 = l_1$

$\langle 2\rangle 1$. $s_l \sqsubseteq s$ and $s_r \sqsubseteq s$
$\langle 3\rangle 1$. Q.E.D.
PROOF: By assumption of $\langle 1\rangle 2$ $s_1 = s_l \sqcup s_r \sqsubseteq s$
$\langle 2\rangle 2$. $l'_a = l_a$ and $s'_l \sqsubseteq s_l$
$\langle 3\rangle 1$. Q.E.D.
PROOF: By induction on derivation depth of $*a$, using $\langle 2\rangle 1$ and assumptions 1 and 2
$\langle 2\rangle 3$. $s'_r \sqsubseteq s_r$
$\langle 3\rangle 1$. Q.E.D.
PROOF: By $\langle 2\rangle 1$ and assumption 1: $\Gamma_2(l_a) \sqsubseteq \Gamma_1(l_a) \sqsubseteq s$
$\langle 2\rangle 4$. Q.E.D.
PROOF: By $\langle 2\rangle 2$ and $\langle 2\rangle 3$, $s_2 = s'_l \sqcup s'_r \sqsubseteq s_l \sqcup s_r = s_1$, and $l_1 = l_2$ by assumption 2 and $\langle 2\rangle 1$.
$\langle 1\rangle 3$. Q.E.D.
PROOF: By induction on the evaluation of l-values and $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

Corollary 1 generalizes the result of Lemma 2 to include all expressions.

Corollary 1 (R-value evaluation in s-equivalent memories)
For all $E, M_1, M_2, \Gamma_1, \Gamma_2, s \in \mathbb{S}$, such that $\Gamma_2 \sqsubseteq_s \Gamma_1$ and $M_1 \sim^s_{\Gamma_1} M_2$,
for all $a \in Exp$ such that $E \vdash a, M_1, \Gamma_1 \Rightarrow v_1, s_1$ and $E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2$
then
$s_1 \sqsubseteq s$ implies that $v_1 = v_2$ and $s_2 \sqsubseteq s_1$

PROOF:
By induction on r-value evaluations, using lemma 2. □

Theorem 1 proves the soundness of our semantics with respect to TINI.

Theorem 1 (Soundness) *The information flow-extended semantics is sound with respect to termination-insensitive non-interference as defined in 2.*

PROOF:
By induction on the evaluation of instructions.
LET: $E, \Gamma_1, \Gamma_2, M_1, M'_1, M_2, M'_2$ and $\underline{pc}_1, \underline{pc}_2 \in \mathbb{S}$ such that:
$E \vdash c, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1$ and $E \vdash c, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2$.
LET: Let $s \in \mathbb{S}$.

ASSUME: 1. $\underline{pc}_2 \sqsubseteq \underline{pc}_1$
2. $\Gamma_2 \sqsubseteq_s \Gamma_1$
3. $M_1 \sim^s_{\Gamma_1} M_2$

PROVE: 1. $\Gamma'_2 \sqsubseteq_s \Gamma'_1$
2. $M'_1 \sim^s_{\Gamma'_1} M'_2$

$\langle 1 \rangle$1. CASE: Skip
PROOF: By assumptions 2 and 3.

$\langle 1 \rangle$2. CASE: Assign
LET: $l_1, s_1, v_2, s_2, l'_1, s'_1, v'_2, s'_2$ such that the evaluation of instruction $a_1 = a_2$ in both M_1, Γ_1 and M_2, Γ_2 yield:

$$(Assign)\ \frac{E \vdash a_1, M_1, \Gamma_1 \Leftarrow l_1, s_1 \quad E \vdash a_2, M_1, \Gamma_1 \Rightarrow v_2, s_2 \quad s_{l_1} = s_1 \sqcup s_2 \sqcup \underline{pc}_1 \quad M'_1 = M_1[l_1 \hookleftarrow v_2] \quad \Gamma''_1 = \Gamma_1[l_1 \hookleftarrow s_{l_1}] \quad \Gamma'_1 = update(a_1 = a_2, s_1, \Gamma''_1)}{E \vdash a_1 = a_2, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1}$$

$$(Assign)\ \frac{E \vdash a_1, M_2, \Gamma_2 \Leftarrow l'_1, s'_1 \quad E \vdash a_2, M_2, \Gamma_2 \Rightarrow v'_2, s'_2 \quad s'_{l_1} = s'_1 \sqcup s'_2 \sqcup \underline{pc}_2 \quad M'_2 = M_2[l'_1 \hookleftarrow v'_2] \quad \Gamma''_2 = \Gamma_2[l'_1 \hookleftarrow s'_{l_1}] \quad \Gamma'_2 = update(a_1 = a_2, s'_1, \Gamma''_2)}{E \vdash a_1 = a_2, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2}$$

SUFFICES ASSUME: $l \in \Gamma'^{-1}_1(s)$
PROVE: 1. $l \in \Gamma'^{-1}_2(s)$
2. $M'_1(l) = M'_2(l)$

$\langle 2 \rangle$1. CASE: $l \in S_p(a_1 = a_2)$
PROOF SKETCH: In this case, the same location $l_1 = l'_1$ is modified by the assignment $a_1 = a_2$ for both runs. It suffices to consider two cases:

- $l = l_1$: the initial values mapped to l are modified in both runs. Yet, the same values are mapped to l after the execution of the assignment since its output security label is below s
- $l \neq l_1$: only the security label of l is modified by the update operator. Yet, it is still below s for both runs.

$\langle 3 \rangle$1. $l_1 = l'_1$ and $s'_1 \sqsubseteq s$
$\langle 4 \rangle$1. $s_1 \sqsubseteq s$
$\langle 5 \rangle$1. $\Gamma'_1(l) = \Gamma''_1(l) \sqcup s_1$
$\langle 5 \rangle$2. Q.E.D.
PROOF: By assumption of $\langle 1 \rangle$2 ($\Gamma'_1(l) \sqsubseteq s$)
$\langle 4 \rangle$2. Q.E.D.
PROOF: by $\langle 4 \rangle$1, and Lemma 2 on the l-value evaluation of a_1 in both M_1, Γ_1 and M_2, Γ_2, and assumptions 2 and 3.

$\langle 3 \rangle$2. CASE: $l = l_1$
$\langle 4 \rangle$1. $\Gamma'_1(l_1) = s_1 \sqcup s_2 \sqcup \underline{pc}_1 \sqsubseteq s$
$\langle 4 \rangle$2. $v_2 = v'_2$ and $s'_2 \sqsubseteq s$
$\langle 5 \rangle$1. $s_2 \sqsubseteq s$
$\langle 6 \rangle$1. Q.E.D.
PROOF: By $\langle 4 \rangle$1
$\langle 5 \rangle$2. Q.E.D.
PROOF: By $\langle 5 \rangle$1, and Corollary 1 on the r-value evaluation of a_2 in both M_1, Γ_1 and M_2, Γ_2, and assumptions 2 and 3.
$\langle 4 \rangle$3. $\Gamma'_2(l_1) = s'_1 \sqcup s'_2 \sqcup \underline{pc}_2 \sqsubseteq s$
$\langle 5 \rangle$1. $\underline{pc}_2 \sqsubseteq s$
$\langle 6 \rangle$1. Q.E.D.
PROOF: By $\langle 4 \rangle$1 and assumption 1
$\langle 5 \rangle$2. Q.E.D.

PROOF: By $\langle 3\rangle 1$ and $\langle 4\rangle 2$ and $\langle 4\rangle 3$

$\langle 4\rangle 4$. Q.E.D.

PROOF: By $\langle 4\rangle 3$ and $\langle 4\rangle 2$

$\langle 3\rangle 3$. CASE: $l \neq l_1$

$\langle 4\rangle 1$. $\Gamma_1(l) \sqsubseteq s$

$\langle 5\rangle 1$. $\Gamma'_1(l) = \Gamma''_1(l) \sqcup s_1$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By the semantics rule of the execution on M_1, Γ_1 and $\langle 2\rangle 1$

$\langle 5\rangle 2$. $\Gamma''_1(l) = \Gamma_1(l)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 3\rangle 3$ and the semantic rule on M_1, Γ_1

$\langle 5\rangle 3$. Q.E.D.

PROOF: By assumption of $\langle 1\rangle 2$ and $\langle 5\rangle 1$ and $\langle 5\rangle 2$

$\langle 4\rangle 2$. $M'_1(l) = M'_2(l)$

$\langle 5\rangle 1$. $M'_1(l) = M_1(l)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 3\rangle 3$

$\langle 5\rangle 2$. $M'_2(l) = M_2(l)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 3\rangle 3$ and $\langle 3\rangle 1$

$\langle 5\rangle 3$. $M_1(l) = M_2(l)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 4\rangle 1$ and assumptions 3 and 2

$\langle 5\rangle 4$. Q.E.D.

PROOF: By $\langle 5\rangle 1$ and $\langle 5\rangle 2$ and $\langle 5\rangle 3$

$\langle 4\rangle 3$. $\Gamma'_2(l) \sqsubseteq s$

$\langle 5\rangle 1$. $\Gamma_2(l) \sqsubseteq s$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 4\rangle 1$ and assumption 2

$\langle 5\rangle 2$. $\Gamma_2(l) = \Gamma''_2(l)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 3\rangle 3$ and $\langle 3\rangle 1$

$\langle 5\rangle 3$. $\Gamma'_2(l) = \Gamma''_2(l) \sqcup s'_1$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By the update operator of the semantic rule on M_2, Γ_2 and $\langle 2\rangle 1$

$\langle 5\rangle 4$. Q.E.D.

PROOF: By $\langle 5\rangle 1$, $\langle 5\rangle 2$, $\langle 5\rangle 3$ and $\langle 3\rangle 1$

$\langle 4\rangle 4$. Q.E.D.

PROOF: By $\langle 4\rangle 3$ and $\langle 4\rangle 2$

$\langle 2\rangle 2$. CASE: $l \notin S_p(a_1 = a_2)$

PROOF SKETCH: In this case, neither the value nor the security label associated to the location l changes.

$\langle 3\rangle 1$. $\Gamma_1(l) = \Gamma'_1(l) \sqsubseteq s$

$\langle 4\rangle 1$. Q.E.D.

PROOF: By $\langle 2\rangle 2$ and assumption of $\langle 1\rangle 2$

$\langle 3\rangle 2$. $M'_1(l) = M'_2(l)$

$\langle 4\rangle 1$. $M'_1(l) = M_1(l)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By $\langle 2\rangle 2$ since $l_1 \neq l$

⟨4⟩2. $M'_2(l) = M_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨2⟩2 since $l'_1 \neq l$
⟨4⟩3. $M_1(l) = M_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨3⟩1 and assumptions 2 and 3
⟨4⟩4. Q.E.D.
PROOF: By ⟨4⟩3, ⟨4⟩2 and ⟨4⟩1
⟨3⟩3. $\Gamma'_2(l) \sqsubseteq s$
⟨4⟩1. $\Gamma'_2(l) = \Gamma_2(l)$
⟨5⟩1. Q.E.D.
PROOF: By ⟨2⟩2
⟨4⟩2. $\Gamma_2(l) \sqsubseteq s$
⟨5⟩1. Q.E.D.
PROOF: By ⟨3⟩1 and assumption 2
⟨4⟩3. Q.E.D.
PROOF: By ⟨4⟩2 and ⟨4⟩1
⟨3⟩4. Q.E.D.
PROOF: By ⟨3⟩3 and ⟨3⟩2
⟨2⟩3. Q.E.D.
PROOF:By ⟨2⟩1 and ⟨2⟩2
⟨1⟩3. CASE: If_{tt}
LET: v_1, s_1, v_2, s_2 such that the evaluation of instruction $if\ (a)\ c_1\ else\ c_2$ in M_1, Γ_1 yield:

$$(If_{tt})\ \frac{\begin{array}{c} E \vdash a, M_1, \Gamma_1 \Rightarrow v_1, s_1 \quad istrue(v_1) \\ \underline{pc}'_1 = s_1 \sqcup \underline{pc}_1 \quad E \vdash c_1, M_1, \Gamma_1, \underline{pc}'_1 \Rightarrow M'_1, \Gamma''_1 \\ \Gamma'_1 = update(c_2, \underline{pc}'_1, \Gamma''_1) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M_1, \Gamma_1, \underline{pc}_1 \Rightarrow M'_1, \Gamma'_1} \qquad RV\ E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2$$

LET: $l \in Loc$
SUFFICES ASSUME: $\Gamma'_1(l) \sqsubseteq s$
PROVE: 1. $\Gamma'_2(l) \sqsubseteq \Gamma'_1(l)$
2. $M'_1(l) = M'_2(l)$
⟨2⟩1. CASE: $s_1 \sqsubseteq s$
PROOF SKETCH: In this case, both executions in M_1, Γ_1 and M_2, Γ_2 execute instruction c_1. Therefore, an induction on c_1 is sufficient.
⟨3⟩1. $v_2 = v_1$ and $s_2 \sqsubseteq s_1$
⟨3⟩2. Q.E.D.
PROOF: By assumption of ⟨2⟩1 and Corollary 1.
Execution is M_2, Γ_2 yield:

$$If_{tt}\ \frac{\begin{array}{c} E \vdash a, M_2, \Gamma_2 \Rightarrow v_2, s_2 \quad istrue(v_2) \\ pc'_2 = s_2 \sqcup \underline{pc}_2 \quad E \vdash c_1, M_2, \Gamma_2 \underline{pc}'_2 \Rightarrow M'_2, \Gamma''_2 \\ \Gamma'_1 = update(c_2, \underline{pc}'_2, \Gamma''_2) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M_2, \Gamma_2, \underline{pc}_2 \Rightarrow M'_2, \Gamma'_2}$$

⟨3⟩3. $pc'_2 \sqsubseteq \underline{pc}'_1$
⟨4⟩1. Q.E.D.
PROOF: By ⟨3⟩1 and assumption 1 and rules If_{tt} of both executions.
⟨3⟩4. $M'_1 \sim^s_{\Gamma_1} M'_2$ and $\Gamma''_2 \sqsubseteq_s \Gamma''_1$
⟨4⟩1. Q.E.D.

PROOF: By ⟨3⟩3, assumptions 3 and 2 and induction hypothesis on the evaluation of instruction c_1.

⟨3⟩5. $\Gamma'_2 \sqsubseteq_s \Gamma'_1$

LET: $l \in loc$

SUFFICES ASSUME: $\Gamma'_1(l) \sqsubseteq s$

PROVE: $\Gamma'_2(l) \sqsubseteq \Gamma'_1(l)$

⟨4⟩1. CASE: $l \notin S_p(c_2)$

⟨5⟩1. Q.E.D.

PROOF: By ⟨3⟩4, assumption of ⟨4⟩1 and definition of operator update, $\Gamma'_1(l) = \Gamma''_2(l) \sqsubseteq \Gamma''_1(l) = \Gamma'_1(l)$

⟨4⟩2. CASE: $l \in S_p(c_2)$

⟨5⟩1. Q.E.D.

PROOF: By $\Gamma'_2(l) = \underline{pc}'_2 \sqsubseteq \Gamma''_2(l)$ and $\Gamma'_1(l) = \underline{pc}'_1 \sqsubseteq \Gamma''_1(l)$ and ⟨3⟩4 and ⟨3⟩3.

⟨4⟩3. Q.E.D.

PROOF: By cases ⟨4⟩1 and ⟨4⟩2

⟨3⟩6. Q.E.D.

PROOF: By ⟨3⟩5 and ⟨3⟩4.

⟨2⟩2. CASE: $s_1 \not\sqsubseteq s$

PROOF SKETCH: In this case, if the label of location l is below s, that means that location l could not have been assigned neither by instructions c_1, nor by instruction c_2. Otherwise, $\underline{pc}$ would have been propagated to the label of l which would be greater than s_1.

⟨3⟩1. $l \notin S_p(c_2)$

⟨4⟩1. Q.E.D.

PROOF: By assumption of ⟨1⟩3 ($\Gamma'_1(l) \sqsubseteq s$). In fact, $l \in S_p(c_2)$ implies that $s_1 \sqsubseteq \Gamma'_1(l)$ which means $\Gamma'_1(l) \not\sqsubseteq s$.

⟨3⟩2. $l \notin S_P(c_1)$

⟨4⟩1. Q.E.D.

PROOF: If there exists an assignment in c_1 that may write to location l, then the update operator would propagate $\underline{pc}'_1 \not\sqsubseteq s$ to $\Gamma'_1(l)$.

⟨3⟩3. $M_1(l) = M'_1(l)$ and $\Gamma_1(l) = \Gamma'_1(l)$

⟨4⟩1. Q.E.D.

PROOF: By ⟨3⟩1 and ⟨3⟩2 since l is neither written by c_1 nor operator update.

⟨3⟩4. $M_2(l) = M'_2(l)$ and $\Gamma_2(l) = \Gamma'_2(l)$

⟨4⟩1. Q.E.D.

PROOF: By ⟨3⟩1 and ⟨3⟩2 since l is neither written by c_1, nor c_2.

⟨3⟩5. Q.E.D.

PROOF: By ⟨3⟩3 and ⟨3⟩4 and assumptions 2 and 3.

⟨2⟩3. Q.E.D.

PROOF: By ⟨2⟩1 and ⟨2⟩2.

⟨1⟩4. CASE: If_{ff}

PROOF: by symmetry of ⟨1⟩3.

⟨1⟩5. CASE: W_{tt} and W_{ff}

⟨2⟩1. Q.E.D.

PROOF: *While* rule is semantically equivalent to $if\ (a)\ c; while\ (a)\ c;\ else\ skip$

⟨1⟩6. CASE: Composition

⟨2⟩1. Q.E.D.

PROOF: By induction on c_1, then on c_2.

⟨1⟩7. Q.E.D.

PROOF: By ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6 and induction on instruction evaluation $\Rightarrow$.

Theorem 2 proves that our program transformation preserves the behavior of the initial program.

Theorem 2 (Initial semantics preservation) *For all* $c, E, M, \Gamma, \underline{pc}, pc$ *such that:*
$E|_{Var(P)} \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1, \Gamma_1$ *and* $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$.

$$\textit{Then, } M_2|_{Loc(P)} = M_1 \textit{ and } \Gamma_2|_{Loc(P)} = \Gamma_1.$$

Proof sketch: By induction on instructions evaluation $\Rightarrow$, knowing that the instructions added by transformation T handle only shadow variables.
Let: $E, M, \Gamma, \underline{pc}, M_1, \Gamma_1, M_2, \Gamma_2$.
Assume: 1. $E \vdash c, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M'_1, \Gamma'_1$
2. $E \vdash T[c, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M'_2, \Gamma'_2$
Prove: 1. $M'_2|_{Loc(P)} = M'_1$
2. $\Gamma'_2|_{Loc(P)} = \Gamma'_1$
$\langle 1 \rangle 1$. Locations pointed by shadow variables are disjoint from initial locations $Loc(P)$
Proof:
Let:
$Loc_M(T[P]) \triangleq \{l : \forall x \in Var(P) \wedge \forall k \in [0, \mathscr{D}(x)] \wedge \forall r \in [0, \mathscr{D}(k)], E \vdash *^r\Lambda(x,k), M \Leftarrow l)\}$.
Then $Loc_M(T[P]) \cap Loc(P) = \emptyset$ since for any $x, y \in Var(P)$, $*^n y$ has a type $ptr^{(*)}(\kappa)$ whereas $*^r\Lambda(x,k)$ has a type $ptr^{(*)}(\tau_s)$. Hence $*^n y$ and $*^r\Lambda(x,k)$ cannot point to the same location l. That stems for the fact that the transformed program is typable if the initial program is typable.
$\langle 1 \rangle 2$. Case: Skip
$\langle 2 \rangle 1$. Q.E.D.
Proof: Holds since output memories are equal to input memories.
$\langle 1 \rangle 3$. Case: Assign
Evaluation of instruction $a_1 = a_2$ in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$ yield :

$$(\textit{Assign})\ \frac{\begin{array}{c} E \vdash a_1, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Leftarrow l_1, s_1 \qquad E \vdash a_2, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v_2, s_2 \\ s = s_1 \sqcup s_2 \sqcup \underline{pc} \qquad s' = s_1 \sqcup \underline{pc} \\ M'_1 = M|_{Loc(P)}[l_1 \mapsto v_2] \qquad \Gamma''_1 = \Gamma|_{Loc(P)}[l_1 \mapsto s] \\ \Gamma'_1 = update(a_1 = a_2, s', \Gamma''_1) \end{array}}{E \vdash a_1 = a_2, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M'_1, \Gamma'_1}$$

Transformation T maps to $a_1 = a_2$ instructions :

$$T[a_1 = a_2, \underline{pc}] \mapsto \begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup \underline{pc}; \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup \underline{pc}; \forall l \in S_P(a_1 = a_2) \\ a_1 = a_2; \end{cases}$$

Let: c_T be the instructions added by the transformation T, and M_T, Γ_T such that: $E \vdash c_T, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$ and $E \vdash a_1 = a_2, M_T, \Gamma_T, \underline{pc} \Rightarrow M'_2, \Gamma'_2$.
$\langle 2 \rangle 1$. $M_T|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma_T|_{Loc(P)} = M_T|_{Loc(P)}$
$\langle 3 \rangle 1$. Q.E.D.
Proof: By $\langle 1 \rangle 1$, c_T does not modify neither values nor security labels mapped by M, Γ for locations $l \in Loc(P)$. Hence, output memories M_T, Γ_T are equal to initial memories M, Γ when both pairs are restricted to the set of initial locations $Loc(P)$.
$\langle 2 \rangle 2$. Q.E.D.
Proof: By $\langle 2 \rangle 1$, and since instruction $a_1 = a_2$ handles only locations in $Loc(P)$, output memories are equal when restricted to $Loc(P)$.

⟨1⟩4. Case: If_{tt}

Supposing that the conditional guard is evaluated to true, evaluation of instruction $if\ (a)\ c_1\ else\ c_2$ in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$ yield:

$$(If_{tt})\ \frac{\begin{array}{c} E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \qquad istrue(v) \\ \underline{pc}' = s \sqcup \underline{pc} \qquad E \vdash c_1, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc}' \Rightarrow M_1, \Gamma_1 \\ \Gamma_1' = update(c_2, \underline{pc}', \Gamma_1) \end{array}}{E \vdash if\ (a)\ c_1\ else\ c_2, M|_{Loc(P)}, \Gamma|_{Loc(P)}, \underline{pc} \Rightarrow M_1', \Gamma_1'}$$

Transformation T maps the initial conditional to the following instructions :

$$T[if\ (a)\ c_1\ else\ c_2, \underline{pc}] \mapsto \begin{cases} \underline{pc}' = \mathcal{L}_R(a) \sqcup \underline{pc}; \\ if\ (a)\ \{ \\ \quad T[c_1, \underline{pc}'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc}'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, \underline{pc}']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc}'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

The same branch is executed for both runs since a is evaluated to the same value (assumed to be $true$).

Let: M_T, Γ_T such that $E \vdash T[c_1, \underline{pc}'], M, \Gamma, \underline{pc}' \Rightarrow M_T, \Gamma_T$.

⟨2⟩1. $M_1'|_{Loc(P)} = M_T$ and $\Gamma_1'|_{Loc(P)} = \Gamma_T$

⟨3⟩1. Q.E.D.

Proof: By induction on the evaluation of both $T[c_1, \underline{pc}']$ in M, Γ and also c_1 in $M|_{Loc(P)}, \Gamma|_{Loc(P)}$.

⟨2⟩2. $M_2'|_{Loc(P)} = M_T|_{Loc(P)}$ and $\Gamma_2'|_{Loc(P)} = \Gamma_T|_{Loc(P)}$

⟨3⟩1. Q.E.D.

Proof: By ⟨1⟩1, instructions $\Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc}'; \forall l \in S_P(c_2)$ and operator $update$ only handle shadow variables. Hence output memories M_2', Γ_2' are equal to input memories M_T, Γ_T when restricted to $Loc(P)$.

⟨2⟩3. Q.E.D.

Proof:By transitivity and ⟨2⟩2 and ⟨2⟩1.

⟨1⟩5. Case: if_{ff}

⟨2⟩1. Q.E.D.

Proof: This case is symmetrical to if_{tt}

⟨1⟩6. Case: W_{tt}

Evaluation of $while\ (a)\ c$ yields:

$$(W_{tt})\ \frac{\begin{array}{c} E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \qquad istrue(v) \qquad \underline{pc}' = s \sqcup \underline{pc} \\ E \vdash c, M, \Gamma, \underline{pc}' \Rightarrow M_1'', \Gamma_1'' \\ E \vdash while\ (a)\ c, M_1'', \Gamma_1'', \underline{pc} \Rightarrow M_1', \Gamma_1 s' \end{array}}{E \vdash while\ (a)\ c, M|_{Loc(P)}, \Gamma, \underline{pc} \Rightarrow M_1', \Gamma_1'}$$

Transformation T yields the following instructions :

$$T[while\ (a)\ c, \underline{pc}] \mapsto \begin{cases} while\ (a)\ \{ \\ \quad \underline{pc}' = \mathcal{L}_R(a) \sqcup \underline{pc}; \\ \quad T[c, \underline{pc}']; \} \\ \underline{pc}' = \mathcal{L}_R(a) \sqcup \underline{pc}; \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc}'; \forall l \in S_P(c) \end{cases}$$

Let: $M_T, \Gamma_T, M_T', \Gamma_T'$ such that :

$E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$, and
$E \vdash \overline{T}[c, \underline{pc'}]; M_T, \Gamma_T, \underline{pc'} \Rightarrow M'_T, \Gamma'_T$, and
$E \vdash T[\overline{while}\ (a)\ c, \underline{pc}], M'_T, \Gamma'_T, \underline{pc} \Rightarrow M'_2, \Gamma'_2$. Then,

$$W_{tt}\ \frac{\begin{array}{c} E \vdash a, M, \Gamma \Rightarrow v, s \qquad istrue(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \\ E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];, M, \Gamma, \underline{pc} \Rightarrow M'_T, \Gamma'_T \\ E \vdash T[\overline{while}\ (a)\ c, \underline{pc}], M'_T, \Gamma'_T, \underline{pc} \Rightarrow M'_2, \Gamma'_2 \end{array}}{E \vdash T[while\ (a)\ c, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M'_2, \Gamma'_2}$$

⟨2⟩1. $M_T|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma_T|_{Loc(P)} = \Gamma|_{Loc(P)}$
⟨3⟩1. Q.E.D.
PROOF: By ⟨1⟩1, knowing that $\underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc}$ handles only shadow variables.
⟨2⟩2. $M'_T|_{Loc(P)} = M''_1$ and $\Gamma'_T|_{Loc(P)} = \Gamma''_1$
⟨3⟩1. Q.E.D.
PROOF: By ⟨2⟩1 and induction on evaluation of both c and $T[c, \underline{pc'}]$.
⟨2⟩3. Q.E.D.
PROOF: By ⟨2⟩2 and induction on both evaluations of $T[while\ (a)\ c, \underline{pc}]$ in M'_T, Γ'_T and $while\ (a)\ c$ in M''_1, Γ''_1.
⟨1⟩7. CASE: W_{ff}
Evaluation of $while\ (a)\ c$ yields :

$$(W_{ff})\ \frac{\begin{array}{c} E \vdash a, M|_{Loc(P)}, \Gamma|_{Loc(P)} \Rightarrow v, s \\ isfalse(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \qquad \Gamma'_1 = update(c, \underline{pc'}, \Gamma|_{Loc(P)}) \end{array}}{E \vdash while\ (a)\ c, M, \Gamma, \underline{pc} \Rightarrow M|_{Loc(P)}, \Gamma'_1}$$

Evaluation of $while\ (a)\ \{E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];\}$ yields :

$$\frac{E \vdash a, M, \Gamma \Rightarrow v, s \qquad isfalse(v) \qquad \underline{pc'} = s \sqcup \underline{pc} \qquad \Gamma''_2 = update(c, \underline{pc'}, \Gamma)}{E \vdash while\ (a)\ \{E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc};\ T[c, \underline{pc'}];\}, M, \Gamma, \underline{pc} \Rightarrow M, \Gamma''_2}$$

Then, evaluation of the remaining instructions yields :
$E \vdash \underline{pc'} = \mathcal{L}_R(a) \sqcup \underline{pc}; \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \underline{pc'}; \forall l \in S_P(c), M, \Gamma''_2, \underline{pc} \Rightarrow M'_2, \Gamma'_2$
⟨2⟩1. $\Gamma'_1 = \Gamma''_2|_{Loc(P)}$
⟨3⟩1. Q.E.D.
PROOF: Operator *update* modifies only the values mapped to locations of $S_P(c)$ in Γ and $\Gamma|_{Loc(P)}$ by assigning to them the same values $\Gamma(l) \sqcup \underline{pc'}, \forall l \in S_P(c)$.
⟨2⟩2. Q.E.D.
PROOF: By ⟨1⟩1 and ⟨2⟩1, $M'_2|_{Loc(P)} = M|_{Loc(P)}$ and $\Gamma'_2|_{Loc(P)} = \Gamma'_1$.
⟨1⟩8. CASE: Composition
⟨2⟩1. Q.E.D.
PROOF: By Induction on both c_1 and c_2
⟨1⟩9. Q.E.D.
PROOF: By ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6, ⟨1⟩7, ⟨1⟩8 and induction on the evaluation rule $\Rightarrow$.

Lemma 1 proves that our program transformation maintains the aliasing invariant.

Lemma 1 (Aliasing invariant)
For all $E, c, M, M', \Gamma, \Gamma', \underline{pc}, pc$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.

Let the predicate $\Omega(M) \triangleq \forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$x \sim^M_{lval} *^r y$$
$$\iff \forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^M_{lval} \Lambda(*^r y, k)$$

Then $\Omega(M) \implies \Omega(M')$.

PROOF SKETCH: By induction on instructions evaluation $\Rightarrow$.
LET: $c, E, M, \Gamma, \underline{pc}, M', \Gamma', pc$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
LET: $\Omega(M) \triangleq \forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$x \sim^M_{lval} *^r y \quad (\mathbf{1})$$
$$\iff \forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^M_{lval} \Lambda(*^r y, k) \quad (\mathbf{2})$$
$$\iff \exists k \geq 0, \Lambda(x,k) \sim^M_{lval} \Lambda(*^r y, k) \quad (\mathbf{3})$$

ASSUME: $\Omega(M)$.
PROVE: $\Omega(M')$.
$\langle 1 \rangle 1$. CASE: Skip
  $\langle 2 \rangle 1$. Q.E.D.
  PROOF: $M = M'$.
$\langle 1 \rangle 2$. CASE: Composition
  $\langle 2 \rangle 1$. Q.E.D.
  PROOF: By induction, M' holds after evaluation of $T[c_1, pc]$. Hence, by induction on $T[c_2, pc]$ $\Omega(M')$ holds after execution of $T[c_1; c_2, pc]$.
$\langle 1 \rangle 3$. CASE: If_{tt}
  Transformation T maps the following instructions to conditionals :

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto \begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

  LET: M_T, Γ_T such that : $E \vdash pc' = \mathcal{L}_R(a) \sqcup pc, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$.
  LET: M_1, Γ_1 such that $E \vdash T[c_1, pc'], M_T, \Gamma_T, \underline{pc'} \Rightarrow M_1, \Gamma_1$.
  LET: M'_1, Γ'_1 such that $E \vdash \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2), M_1, \Gamma_1, \underline{pc'} \Rightarrow M'_1, \Gamma'_1$.
  LET: M', Γ' such that $E \vdash if(a)\{T[c_1, pc']; \ldots;\}\ else\ \{\ldots\}, M_T, \Gamma_T, \underline{pc'} \Rightarrow M', \Gamma'$.
  $\langle 2 \rangle 1$. $\Omega(M_T)$ holds
    $\langle 3 \rangle 1$. Q.E.D.
    PROOF: $pc' = \mathcal{L}_R(a) \sqcup pc$; only modifies pc', which is of type τ_s. No l-values are modified, hence $\Omega(M_T)$ holds since $\Omega(M)$ holds.
  $\langle 2 \rangle 2$. $\Omega(M_1)$ holds
    $\langle 3 \rangle 1$. Q.E.D.
    PROOF: By induction on $T[c_1, pc']$ knowing $\Omega(M_T)$.
  $\langle 2 \rangle 3$. $\Omega(M'_1)$ holds
    $\langle 3 \rangle 1$. Q.E.D.
    PROOF: Modified variables are of type τ_s, therefore no l-values are modified. Hence $\Omega(M'_1)$ holds from $\Omega(M_1)$.
  $\langle 2 \rangle 4$. Q.E.D.
  PROOF: M' is equal to M'_1 by the evaluation rule if_{tt}. Hence $\Omega(M')$ holds by $\langle 2 \rangle 3$.
$\langle 1 \rangle 4$. CASE: if_{ff}
  $\langle 2 \rangle 1$. Q.E.D.
  PROOF: This case is symmetrical to the if_{tt} rule.
$\langle 1 \rangle 5$. CASE: W_{tt} and W_{ff}

⟨2⟩1. Q.E.D.
PROOF: Derives from the cases if_{tt} and if_{ff}.
⟨1⟩6. CASE: Assign
PROOF SKETCH: This is the most interesting case. We are going to prove that new aliasing relations for initial variables are reproduced for shadow variables, without breaking old aliasing relations. Transformation T maps the following instructions to assignment $a_1 = a_2$:

$$T[a_1 = a_2, pc] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; & \text{(instruction } c_T^0\text{)} \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] & \text{(instruction } c_T^1\text{)} \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 = a_2) & \text{(instruction } c_T^2\text{)} \\ a_1 = a_2; \end{cases}$$

LET: instructions c_T^0, c_T^1 and c_T^2 be as shown above, and $c_T = c_T^0; c_T^1; c_T^2;$.
LET: M_1, Γ_1, M_2, Γ_2 and M_3, Γ_3 such that: $E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1$, and
$E \vdash c_T^1, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$, and $E \vdash c_T^2, M, \Gamma, \underline{pc} \Rightarrow M_3, \Gamma_3$.
Then (only relevant semantics operations are shown below) :

$$(Assign)\ \frac{\begin{array}{c} E \vdash \Lambda(a_1, 0), M, \Gamma \Leftarrow l_{\Lambda(a_1,0)}, s_{\Lambda(a_1,0)} \\ E \vdash \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc, M, \Gamma \Rightarrow v_{\Lambda(a_1,0)}, s'_{\Lambda(a_1,0)} \qquad M_1 = M[l_1 \mapsto v_2] \qquad \ldots \end{array}}{E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1}$$

$$E \vdash c_T^1, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2 \qquad\qquad E \vdash c_T^2, M_2, \Gamma_2, \underline{pc} \Rightarrow M_3, \Gamma_3$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_3, \Gamma_3 \Leftarrow l_{a_1}, s_{a_1} \\ E \vdash a_2, M_3, \Gamma_3 \Rightarrow s_{a_2}, v_{a_2} \qquad M' = M_3[l_{a_1} \mapsto v_{a_2}] \qquad \ldots \end{array}}{E \vdash a_1 = a_2, M_3, \Gamma_3, \underline{pc} \Rightarrow M', \Gamma'}$$

$$E \vdash T[a_1 = a_2, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

⟨2⟩1. $M_3|_{Loc(P)} = M_2|_{Loc(P)} = M_1|_{Loc(P)} = M|_{Loc(P)}$
⟨3⟩1. Q.E.D.
PROOF: Since locations pointed by initial variables are disjoint from locations pointed by shadow variables, and c_T only handles shadow variables (this have been proven in previous lemma).
⟨2⟩2. CASE: $\Omega(M').(1) \implies \Omega(M').(2)$
ASSUME: $\Omega(M)$ and $\forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$x \sim_{lval}^{M'} *^r y \quad (1)$$

$$\implies \forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim_{lval}^{M'} \Lambda(*^r y, k) \quad (2)$$

PROOF SKETCH: By induction on $r \in \mathbb{N}$. The special case when $r = 1$ is also proven in order to be used later during the proof.
⟨3⟩1. CASE: $r = 0$
⟨4⟩1. Q.E.D.
PROOF: $x \sim_{lval}^{M'} y$ implies x and y are the same variable since environment E is a bijection.
⟨3⟩2. CASE: $r = 1$
ASSUME: $x \sim_{lval}^{M'} *y$
PROVE: $\forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim_{lval}^{M'} \Lambda(*y, k)$
LET: $l_y \in Loc(P)$ such that $E \vdash y, M' \Leftarrow l_y$ and $M'[l_y] = l_x$ (We will explicit neither Γ nor security labels as they are useless for this). Notice that $E \vdash y, M \Leftarrow l_y$ since

$E(y) = l_y$.

PROOF SKETCH: Either a_1 is an alias for y (in both M or M'), then a_1 modifies the r-value of y, and T create new aliasing relations for shadow variables of x and $*y$. Or a_1 is not an alias for y, then the aliasing relations already exist in M and are not modified in M'.

$\langle 4\rangle 1$. CASE: $a_1 \sim^{M'}_{lval} y$

$\langle 5\rangle 1$. $a_1 \sim^{M}_{lval} y$

$\langle 6\rangle 1$. Q.E.D.

PROOF: $M_3|_{Loc(P)} = M|_{Loc(P)}$ ($\langle 2\rangle 1$) and $a_1 = a_2$ modifies only the r-value of a_1, not its l-value.

$\langle 5\rangle 2$. $*a_2 \sim^{M}_{lval} x$

$\langle 6\rangle 1$. Q.E.D.

PROOF: $a_1 = a_2$ maps v_2 to l_{a_1} in M'. As $*a_1$ end up being aliased to $\&x$ in M' (assumption of $\langle 3\rangle 2$), v_2 must be equal to $ptr(l_x)$ in M_3. Hence $*a_2$ is an alias for x in M_3, and also in M by $\langle 2\rangle 1$.

$\langle 5\rangle 3$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_2, k) \sim^{M}_{lval} \Lambda(x, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 5\rangle 2$ and $\Omega(M)$.

$\langle 5\rangle 4$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_2, k) \sim^{M'}_{lval} \Lambda(x, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: The l-values of $\Lambda(*a_2, k) = *\Lambda(a_2, k+1)$ in M are equal to those in M'. If the r-value of $\Lambda(a_2, k+1)$ were to change, it would definitely be because of c^1_T (the only instructions manipulations pointer shadow variables). That would mean that there exists k_1 such that $\Lambda(a_1, k_1) \sim^{M}_{lval} \Lambda(a_2, k+1)$ (). That would imply that $k_1 = k+1$ by typing. Hence, $\Omega(M)$ implies that a_1 and a_2 are aliased, and so $\Lambda(a_2, k) \sim^{M}_{lval} \Lambda(a_1, k)$ for all $k \in [0, \mathscr{D}(a_1)]$, meaning that c_T overrides the r-values of $\Lambda(a_2, k+1)$ in M by writing the same r-value in M'.

$\langle 5\rangle 5$. $\forall k \in [0, \mathscr{D}(y)], \Lambda(a_1, k) \sim^{M}_{lval} \Lambda(y, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\langle 5\rangle 1$ and $\Omega(M)$.

$\langle 5\rangle 6$. $\forall k \in [0, \mathscr{D}(y)], \Lambda(a_1, k) \sim^{M'}_{lval} \Lambda(y, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: Only r-values of $\Lambda(a_1, k)$ are modified by c_T, and the l-value of $\Lambda(y, k) = E(\Lambda(y, k))$ is constant.

$\langle 5\rangle 7$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(*a_1, k) \sim^{M'}_{lval} \Lambda(*a_2, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: After the assignments c_T, for all $k \in [1, \mathscr{D}(a_2)]$, the r-values of $\Lambda(a_1, k)$ are equal to $\Lambda(a_2, k)$. Dereferencing both means that $\forall k \in [1, \mathscr{D}(a_2)]$, we have $*\Lambda(a_1, k) \sim^{M'}_{lval} *\Lambda(a_2, k)$. Hence $\forall k \in [1, \mathscr{D}(a_2)], \Lambda(*a_1, k-1) \sim^{M'}_{lval} \Lambda(*a_2, k-1)$. A change of variable $k \mapsto k-1$ gives the desired result, knowing that $\mathscr{D}(a_2) = \mathscr{D}(x)+1$ by $\langle 5\rangle 2$ and typing.

$\langle 5\rangle 8$. $\forall k \in [0, \mathscr{D}(*y)], \Lambda(*a_1, k) \sim^{M'}_{lval} \Lambda(*y, k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By dereferencing $\langle 5\rangle 6$ for $k \in [1, \mathscr{D}(y)]$ and a variable change $k \mapsto k-1$.

$\langle 5\rangle 9$. Q.E.D.

PROOF: By transitivity and $\langle 5\rangle 8$, $\langle 5\rangle 7$ and $\langle 5\rangle 4$, $\forall k \in [0, \mathscr{D}(x)], \Lambda(x, k) \sim^{M'}_{lval} \Lambda(*y, k)$.

$\langle 4\rangle 2$. CASE: $\neg(a_1 \sim^{M'}_{lval} y)$

$\langle 5\rangle 1$. $x \sim^{M}_{lval} y$

$\langle 6\rangle 1$. Q.E.D.

PROOF: The r-value of y is not changed by assignment $a_1 = a_2$ since $\langle 4\rangle 2$. It is neither changed by c_T since $\langle 2\rangle 1$.

$\langle 5\rangle 2$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M}_{lval} \Lambda(*y,k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: By $\Omega(M)$.

$\langle 5\rangle 3$. $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*y,k)$

$\langle 6\rangle 1$. Q.E.D.

PROOF: If exists k such that the l-value of $\Lambda(*y,k) = *\Lambda(y,k+1)$ is changed by c_T, then the r-value of $\Lambda(y,k+1)$ is overridden by c_T. That means that there exists k_1 such that $\Lambda(a_1,k_1) \sim^{M}_{lval} \Lambda(y,k+1)$. Hence, by typing $k_1 = k+1$ and $\Lambda(a_1,k_1) \sim^{M}_{lval} \Lambda(y,k_1)$. Then $\Omega(M)$ implies that $a_1 \sim^{M}_{lval} y$ which was supposed to not hold in this case $\langle 4\rangle 2$.

$\langle 5\rangle 4$. Q.E.D.

PROOF: By $\langle 5\rangle 3$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 1$ and $\langle 4\rangle 2$.

$\langle 3\rangle 3$. CASE: $r \geq 1$

ASSUME: $x \sim^{M'}_{lval} *^{r+1}y$

PROVE: $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y,k)$

$\langle 4\rangle 1$. $\exists z \in Var(P)$, such that $*^r y \sim^{M'}_{lval} z$ and $\forall k \in [0, \mathscr{D}(z)], \Lambda(z,k) \sim^{M'}_{lval} \Lambda(*^r y,k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: Since the only way to create locations is through variable declaration, there exists a variable z which is pointed by $*^r y$. By induction on r, we have $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^r y,k)$.

$\langle 4\rangle 2$. $\forall k \in [0, \mathscr{D}(*z)], \Lambda(*z,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y,k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By dereferencing one time both sides of the aliasing relation in $\langle 4\rangle 1$ for $k \in [1, \mathscr{D}(z)]$ and a variable change $k \mapsto k-1$.

$\langle 4\rangle 3$. $x \sim^{M'}_{lval} *z$ and $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*z,k)$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By $\langle 4\rangle 1$, we have $x \sim^{M'}_{lval} *z$. Then by the case $r = 1$ ($\langle 3\rangle 2$), we have the aliasing relation for shadow variables.

$\langle 4\rangle 4$. Q.E.D.

PROOF: By $\langle 4\rangle 3$ and $\langle 4\rangle 2$ and transitivity, $\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M'}_{lval} \Lambda(*^{r+1}y,k)$.

$\langle 3\rangle 4$. Q.E.D.

PROOF: By induction on r, $\langle 3\rangle 3$ and $\langle 3\rangle 1$.

$\langle 2\rangle 3$. CASE: $\Omega(M').(3) \iff \Omega(M').(2) \implies \Omega(M').(1)$

ASSUME: $\Omega(M)$ and $\forall x, y \in Var(P)$, for all $r \in [0, \mathscr{D}(y)]$,

$$\forall k \in [0, \mathscr{D}(x)], \Lambda(x,k) \sim^{M}_{lval} \Lambda(*^r y,k) \quad (2)$$
$$\iff \exists k \geq 0, \Lambda(x,k) \sim^{M}_{lval} \Lambda(*^r y,k) \quad (3)$$
$$\implies x \sim^{M}_{lval} *^r y \quad (1)$$

PROOF SKETCH: (2) $\implies$ (3) holds. We prove (3) $\implies$ (2) by induction on r as in $\langle 2\rangle 2$. We focus on the most interesting case, $r = 1$.

$\langle 3\rangle 1$. CASE: $r = 1$

ASSUME: $x, y \in Var(P)$, such that $\exists k_1, \Lambda(x,k_1) \sim^{M'}_{lval} \Lambda(*y,k_1)$.

$\langle 4\rangle 1$. CASE: $\exists k'$ such that the l-value of $\Lambda(*y,k')$ changed between M and M'

$\langle 5\rangle 1$. $a_1 \sim^{M}_{lval} y$

$\langle 6\rangle 1$. Q.E.D.

PROOF: The l-value of $\Lambda(*y,k') = *\Lambda(y,k'+1)$ changed, meaning that the r-value

of $\Lambda(y, k'+1)$ has been overridden by c_T. That means that there exists k_1 such that $\Lambda(a_1, k_1) \sim^M_{lval} \Lambda(y, k'+1)$ and $k'+1 = k_1$. Then by $\Omega(M)$, $a_1 \sim^M_{lval} y$.

$\langle 5\rangle 2$. $x \sim^M_{lval} *a_2$

$\langle 6\rangle 1$. Q.E.D.

PROOF: Since a_2 has been assigned to a_1, and $a_1 \sim^M_{lval} y$ ($\langle 5\rangle 1$), the r-value of $\Lambda(a_2, k_1+1)$ in M has been assigned to $\Lambda(y, k_1)+1)$ in M'. Knowing $\Lambda(y, k_1+1)$ in M' points to $\Lambda(x, k_1)$, we conclude that $\Lambda(a_2, k_1+1)$ in M' points to $\Lambda(x, k_1)$. Hence $*\Lambda(a_2, k_1+1) \sim^M_{lval} \Lambda(x, k_1)$. Then by $\Omega(M)$ and $\Lambda(*a_2, k_1) \sim^M_{lval} \Lambda(x, k_1)$, we conclude that $x \sim^M_{lval} *a_2$.

$\langle 5\rangle 3$. Q.E.D.

PROOF: For all $0 \leq k \leq \mathscr{D}(a_1) - 1$, the r-value of $\Lambda(a_1, k+1)$ in M' is equal to that of $\Lambda(a_2, k+1)$ in M after evaluation of assignments c_T. By $\langle 5\rangle 2$, $\Lambda(a_2, k+1)$ points to $\Lambda(x, k)$. Therefore, $\langle 5\rangle 1$, $\Lambda(y, k+1)$ also points to $\Lambda(x, k)$, meaning that $\Lambda(*y, k) \sim^{M'}_{lval} \Lambda(x, k)$, for all $k \in [0, \mathscr{D}(x)]$. Also, by $\langle 5\rangle 1$ and $\langle 5\rangle 2$, $x \sim^{M'}_{lval} *y$ holds after evaluation of $a_1 = a_2$.

$\langle 4\rangle 2$. CASE: $\neg(\exists k'$ such that the l-value of $\Lambda(*y, k')$ changed between M and $M')$

$\langle 5\rangle 1$. Q.E.D.

PROOF: $\forall k \in [0, \mathscr{D}(*y)]$, the l-value of $\Lambda(*y, k')$ in M is equal to that in M'. Particularly, $\Lambda(*y, k_1) \sim^M_{lval} \Lambda(x, k_1)$. By $\Omega(M)$, $\forall k \in [0, \mathscr{D}(x)], \Lambda(*y, k_1) \sim^M_{lval} \Lambda(x, k)$. Which means that $\forall k \in [0, \mathscr{D}(x)], \Lambda(*y, k_1) \sim^{M'}_{lval} \Lambda(x, k)$ since neither l-value of $\Lambda(*y, k_1)$ changed. Also, By $\Omega(M)$, we have $x \sim^M_{lval} *y$. Hence, $x \sim^{M'}_{lval} *y$ since if the r-value of y changes, that would mean that a_1 is an alias for y, causing the r-value of $\Lambda(*y, k)$ to be overridden which is not the case since $\langle 4\rangle 2$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 2$ and $\langle 4\rangle 1$.

$\langle 3\rangle 2$. Q.E.D.

PROOF: Using the case $r = 1$ ($\langle 3\rangle 1$), we conclude by introducing a variable z such that $*z \sim^{M'}_{lval} x$ and then by induction on r as done in $\langle 2\rangle 2$.

Theorem 3 proves that our program transformation is sound with respect to our monitor semantics.

Theorem 3 (Sound monitoring of information flows) *Let c, for all $E, M, \Gamma, M', \Gamma'$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.*

Let us define the predicate $\Upsilon(E, M, \Gamma) \triangleq$ for all $x \in Var(P)$, for all $k \in [0, \mathscr{D}(x)]$, $E \vdash *^k x, M \Leftarrow l_{xk}$ and $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}$.

$$\textit{The following result holds: } \Upsilon(E, M, \Gamma) \textit{ and } E \vdash pc, M \Rightarrow \underline{pc} \implies \Upsilon(E, M', \Gamma').$$

First, we prove that operators $\mathcal{L}_L$ and $\mathcal{L}_R$ captures resp. the labels of l-value evaluation and r-value evaluation. We introduce hence Lemma 3 and Corollary 2.

Lemma 3 (Sound operator $\mathcal{L}_L$) *Let E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds. for all $a \in Exp$,*

$$E \vdash a, M, \Gamma \Leftarrow l, s \implies E \vdash \mathcal{L}_L(a), M \Rightarrow s$$

PROOF SKETCH:By induction on l-value evaluations.
LET: E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds.
LET: $a \in Exp, l \in Loc, s \in \mathbb{S}$, such that $E \vdash a, M, \Gamma \Leftarrow l, s$.
PROVE: $E \vdash \mathcal{L}_L(a), M \Rightarrow s$

⟨1⟩1. Case: LV_{ID}
⟨2⟩1. Q.E.D.
Proof: By definition of $\mathcal{L}_L(id)$ and rule LV_{ID}, both are equal to *public*
⟨1⟩2. Case: LV_{MEM}
Evaluation of $*^k x$ in l-value position yields:

$$LV_{MEM}\ \dfrac{RV\ \dfrac{E \vdash a, M, \Gamma \Leftarrow l_a, s_l \qquad M(l_a) = ptr(l) \qquad \Gamma(l_a) = s_r \qquad s = s_l \sqcup s_r}{E \vdash a, M, \Gamma \Rightarrow ptr(l), s}}{E \vdash *a, M, \Gamma \Leftarrow l, s}$$

Also, by definition we have: $\mathcal{L}_L(*a) = \mathcal{L}_R(a) = \mathcal{L}_L(a) \sqcup \mathcal{L}(a)$.
⟨1⟩3. Q.E.D.
Proof: By induction, we have $E \vdash \mathcal{L}_L(a), M \Rightarrow s_l$. Also, $\Upsilon(E, M, \Gamma)$ implies that $E \vdash \mathcal{L}(a), M \Rightarrow s_r$. Hence, $E \vdash \mathcal{L}_L(*a), M \Rightarrow s$.

Corollary 2 (Sound operator $\mathcal{L}_R$) *Let E, M, Γ, such that $\Upsilon(E, M, \Gamma)$ holds. for all $a \in Exp$,*

$$E \vdash a, M, \Gamma \Rightarrow v, s \implies E \vdash \mathcal{L}_R(a), M \Rightarrow s$$

⟨1⟩1. Q.E.D.
Proof: By induction on r-value evaluations of expressions, using Lemma 3.

Theorem 3 proves that our program transformation is sound with respect to our monitor semantics.

Theorem 3 (Sound monitoring of information flows) *Let c, for all $E, M, \Gamma, M', \Gamma'$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.*

Let us define the predicate $\Upsilon(E, M, \Gamma) \triangleq$ for all $x \in Var(P)$, for all $k \in [0, \mathscr{D}(x)]$, $E \vdash *^k x, M \Leftarrow l_{xk}$ and $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}$.

The following result holds: $\Upsilon(E, M, \Gamma)$ and $E \vdash pc, M \Rightarrow \underline{pc} \implies \Upsilon(E, M', \Gamma')$.

Proof sketch: By induction on instructions evaluation $\Rightarrow$. The most interesting case is the assignment.
Let: $c, E, M, \Gamma, \underline{pc}, M', \Gamma', \underline{pc}$ such that $E \vdash T[c, pc], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$.
Let: $\Upsilon(E, M, \Gamma) \triangleq$ for all $x \in Var(P)$, for all $k \in [0, \mathscr{D}(x)]$,
$E \vdash *^k x, M \Leftarrow l_{xk}$ and $\Gamma(l_{xk}) = s_{xk} \implies E \vdash *^k \Lambda(x, k), M \Rightarrow s_{xk}$.
Assume: $\Upsilon(E, M, \Gamma)$ and $E \vdash pc, M \Rightarrow \underline{pc}$.
Prove: $\Upsilon(E, M', \Gamma')$.
⟨1⟩1. Case: Skip
⟨2⟩1. Q.E.D.
Proof: By assumption.
⟨1⟩2. Case: Assign
Proof sketch: We prove $\Upsilon(E, M', \Gamma')$ for any $x \in Var(P)$. Then we generalize this result by Lemma 1 to every l-value $*^k x$. Transformation T maps the following instructions to assignment $a_1 = a_2$:

$$T[a_1 = a_2, pc] \mapsto$$

$$\begin{cases} \Lambda(a_1, 0) = \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc; & (\text{instruction } c_T^0) \\ \Lambda(a_1, k) = \Lambda(a_2, k); \forall k \in [1, \mathscr{D}(a_1)] & (\text{instruction } c_T^1) \\ \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup \mathcal{L}_L(a_1) \sqcup pc; \forall l \in S_P(a_1 = a_2) & (\text{instruction } c_T^2) \\ a_1 = a_2; \end{cases}$$

LET: instructions c_T^0, c_T^1 and c_T^2 be as shown above, and $c_T = c_T^0; c_T^1; c_T^2;$.
LET: M_1, Γ_1, M_2, Γ_2 and M_3, Γ_3 such that: $E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1$, and
$E \vdash c_T^1, M, \Gamma, \underline{pc} \Rightarrow M_2, \Gamma_2$, and $E \vdash c_T^2, M, \Gamma, \underline{pc} \Rightarrow M_3, \Gamma_3$.
Then (only relevant semantics operations are shown below) :

$$(Assign)\ \frac{\begin{array}{c} E \vdash \Lambda(a_1,0), M, \Gamma \Leftarrow l_{\Lambda(a_1,0)}, s_{\Lambda(a_1,0)} \\ E \vdash \mathcal{L}_L(a_1) \sqcup \mathcal{L}_R(a_2) \sqcup pc, M, \Gamma \Rightarrow v_{\Lambda(a_1,0)}, s'_{\Lambda(a_1,0)} \\ M_1 = M[l_{\Lambda(a_1,0)} \mapsto v_{\Lambda(a_1,0)}] \qquad \ldots \end{array}}{E \vdash c_T^0, M, \Gamma, \underline{pc} \Rightarrow M_1, \Gamma_1} \qquad E \vdash c_T^1, M_1, \Gamma_1, \underline{pc} \Rightarrow M_2, \Gamma_2$$

$$E \vdash c_T^2, M_2, \Gamma_2, \underline{pc} \Rightarrow M_3, \Gamma_3$$

$$(Assign)\ \frac{\begin{array}{c} E \vdash a_1, M_3, \Gamma_3 \Leftarrow l_{a_1}, s_{a_1} \qquad E \vdash a_2, M_3, \Gamma_3 \Rightarrow s_{a_2}, v_{a_2} \\ M' = M_3[l_{a_1} \mapsto v_{a_2}] \qquad s = s_{a_1} \sqcup s_{a_2} \sqcup \underline{pc} \qquad \Gamma'' = \Gamma_3[l_{a_1} \mapsto s] \\ \Gamma' = update(a_1 = a_2, s_{a_1} \sqcup \underline{pc}, \Gamma'') \qquad \ldots \end{array}}{E \vdash a_1 = a_2, M_3, \Gamma_3, \underline{pc} \Rightarrow M', \Gamma'}$$

$$E \vdash T[a_1 = a_2, \underline{pc}], M, \Gamma, \underline{pc} \Rightarrow M', \Gamma'$$

$\langle 2 \rangle 1.\ \forall x \in Var(P), \Gamma'(E(x)) = s \implies E \vdash \Lambda(x,0), M' \Rightarrow s$
LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.
$\langle 3 \rangle 1.$ CASE: $a_1 \sim_{lval}^{M'} x$
PROOF SKETCH: In this case, $T[c, pc]$ updates both x and $\Lambda(x,0)$ such that Υ still holds.
$\langle 4 \rangle 1.\ s = \Gamma'(E(x)) = s_{a_1} \sqcup s_2 \sqcup \underline{pc}$
$\langle 5 \rangle 1.$ Q.E.D.
PROOF: By assignment rule of $a_1 = a_2$ since only instruction $a_1 = a_2$ modifies values mapped to $l_{a_1} = E(x) \in Loc(P)$.
$\langle 4 \rangle 2.\ s = v_{\Lambda(a_1,0)}$
$\langle 5 \rangle 1.$ Q.E.D.
PROOF: By Lemma 1, we have $\Lambda(a_1,0) \sim_{lval}^{M'} \Lambda(x,0)$. Then we conclude by knowing that $\Lambda(a_1,0)$ is assigned by instruction c_T^0. That means that $E \vdash \Lambda(a_1,0), M' \Rightarrow v_{\Lambda(a_1,0)}$. (notice that c_T^2 also assigns $\Lambda(a_1,0)$. But since it just propagates r-values of $\mathcal{L}_L(a_1) \sqcup pc$ which are already propagated by c_T^0, the value of $\Lambda(a_1,0)$ keeps being equal to $v_{\Lambda(a_1,0)}$)
$\langle 4 \rangle 3.$ Q.E.D.
PROOF: By $\langle 4 \rangle 1$, $\langle 4 \rangle 2$ and assumption $\Upsilon(E, M, \Gamma)$, $v_{\Lambda(a_1,0)} = s_{a_1} \sqcup s_2 \sqcup \underline{pc}$ (Lemma 3 and Corollary 2).
$\langle 3 \rangle 2.$ CASE: $\neg(a_1 \sim_{lval}^{M'} x)$ and $E(x) \notin S_P(a_1 = a_2)$
LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.
LET: $v_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x,0), M' \Rightarrow v_{\Lambda(x,0)}$
PROOF SKETCH: In this case, neither x nor $\Lambda(x,0)$ are updated. Hence the invariant holds from $\Upsilon(E, M, \Gamma)$.
$\langle 4 \rangle 1.\ E \vdash \Lambda(x,0), M \Rightarrow v_{\Lambda(x,0)}$
$\langle 5 \rangle 1.$ Q.E.D.
PROOF: Since $\Lambda(x,0)$ in not modified by $T[a_1 = a_2, pc]$. $a_1 = a_2$ modifies only locations in $Loc(P)$. c_T^2 do not modify $\Lambda(x,0)$ since $E(x) \notin S_P(a_1 = a_2)$. c_T^1 modifies only pointers and $\Lambda(x,0)$ is not. c_T^0 does not assign $\Lambda(x,0)$ since $\neg(a_1 \sim_{lval}^{M'} x)$.
$\langle 4 \rangle 2.\ \Gamma(E(x)) = s$
$\langle 5 \rangle 1.$ Q.E.D.
PROOF: By the semantics of assignments, knowing that $E(x) \notin S_P(a_1 = a_2)$.
$\langle 4 \rangle 3.$ Q.E.D.

PROOF:By $\langle 3\rangle 2$ and $\langle 4\rangle 2$ and $\Upsilon(E, M, \Gamma)$, $s = v_{\Lambda(x,0)}$

$\langle 3\rangle 3$. CASE: $\neg(a_1 \sim^{M'}_{lval} x)$ and $E(x) in S_P(a_1 = a_2)$

PROOF SKETCH: In this case, $s_{a_1} \sqcup \underline{pc}$ and $\mathcal{L}_L(a_1)$ are respectively propagated to respectively $\Gamma(E(x))$ and $\Lambda(x, 0)$. We show that r-value evaluation of $\Lambda(x, 0)$ in M' is equal to the value of $\Gamma(l_{a_1})$.

LET: $x \in Var(P)$ such that $\Gamma'(E(x)) = s$.

LET: $v'_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x, 0), M' \Rightarrow v_{\Lambda(x,0)}$.

LET: $v_{\Lambda(x,0)}$ such that $E \vdash \Lambda(x, 0), M \Rightarrow v_{\Lambda(x,0)}$.

$\langle 4\rangle 1$. Q.E.D.

PROOF: By $\Gamma'(E(x)) = s_{a_1} \sqcup \underline{pc} \sqcup \Gamma(E(x))$ and $v'_{\Lambda(x,0)} = \mathcal{L}_L(a_1) \sqcup pc \sqcup v_{\Lambda(x,0)}$. Also, by $\Upsilon(E, M, \Gamma)$ we have $s_{a_1} = \mathcal{L}_L(a_1)$ and $v_{\Lambda(x,0)} = \Gamma(E(x))$.

$\langle 3\rangle 4$. Q.E.D.

PROOF: By $\langle 3\rangle 3$ and $\langle 3\rangle 2$ and $\langle 3\rangle 1$.

$\langle 2\rangle 2$. Q.E.D.

PROOF:By $\langle 2\rangle 1$ we can conclude. For all $y \in Var(P)$, $k \in [0, \mathscr{D}(y)]$, there exists $x \in Var(P)$ such that $x \sim^{M'}_{lval} *^k y$. Then $E(x)$ is the l-value of $*^k y$ in memory M'.

Let $s = \Gamma'(E(x))$. That implies that $E \vdash \Lambda(x, 0), M' \Rightarrow s$ ($\langle 2\rangle 1$), and also $E \vdash *^k \Lambda(y, k), M' \Rightarrow s$ since $\Lambda(*^k y, 0) \sim^{M'}_{lval} \Lambda(x, 0)$ (Lemma 1).

$\langle 1\rangle 3$. CASE: Composition

$\langle 2\rangle 1$. Q.E.D.

By induction on c_1 and then on c_2.

$\langle 1\rangle 4$. CASE: If_{tt}

Transformation T maps the following instructions to conditionals :

$$T[if\ (a)\ c_1\ else\ c_2, pc] \mapsto \begin{cases} pc' = \mathcal{L}_R(a) \sqcup pc; \\ if\ (a)\ \{ \\ \quad T[c_1, pc'] \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2) \\ \}\ else\ \{ \\ \quad T[c_2, pc']; \\ \quad \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_1) \\ \} \end{cases}$$

LET: M_T, Γ_T such that : $E \vdash pc' = \mathcal{L}_R(a) \sqcup pc, M, \Gamma, \underline{pc} \Rightarrow M_T, \Gamma_T$.

LET: M_1, Γ_1 such that $E \vdash T[c_1, pc'], M_T, \Gamma_T, \underline{pc'} \Rightarrow M_1, \Gamma_1$.

LET: M'_1, Γ'_1 such that $E \vdash \Lambda(E^{-1}(l), 0) = \Lambda(E^{-1}(l), 0) \sqcup pc'; \forall l \in S_P(c_2), M_1, \Gamma_1, \underline{pc'} \Rightarrow M'_1, \Gamma'_1$.

LET: M', Γ' such that $E \vdash if(a)\{T[c_1, \underline{pc'}]; \ldots;\}\ else\ \{\ldots\}, M_T, \Gamma_T, \underline{pc'} \Rightarrow M', \Gamma'$.

$\langle 2\rangle 1$. $\Upsilon(E, M_T, \Gamma_T)$ holds and $E \vdash pc', M_T \Rightarrow \underline{pc'}$

$\langle 3\rangle 1$. Q.E.D.

PROOF: Since assignment pc' does not modify neither locations in $Loc(P)$ nor locations associated to shadow variables defined by Λ, $\Upsilon(E, M_T, \Gamma_T)$ holds. Additionally, $\underline{pc'} = s_a \sqcup \underline{pc}$ where s_a is the result of r-value evaluation of a in M and $E \vdash \mathcal{L}_R(a), M \Rightarrow s_a$. Hence $E \vdash pc', M \Rightarrow \underline{pc'}$.

$\langle 2\rangle 2$. $\Upsilon(E, M_1, \Gamma_1)$ holds

$\langle 3\rangle 1$. Q.E.D.

By induction on the execution of $T[c_1, pc']$, knowing $\langle 2\rangle 1$.

$\langle 2\rangle 3$. $\Upsilon(E, M', \Gamma')$

$\langle 3\rangle 1$. Q.E.D.

PROOF: $\underline{pc'}$ is propagated to all shadow variables corresponding to locations in $S_P(c_2)$. So does the update operator on Γ'_1 before yielding Γ'. The result for other locations (not in $S_P(c_2)$) shadow variables stems from $\langle 2 \rangle 2$.

$\langle 2 \rangle 4$. Q.E.D.

PROOF: By $\langle 2 \rangle 3$.

$\langle 1 \rangle 5$. CASE: If_{ff}

$\langle 2 \rangle 1$. Q.E.D.

Symmetrical case to If_{tt}.

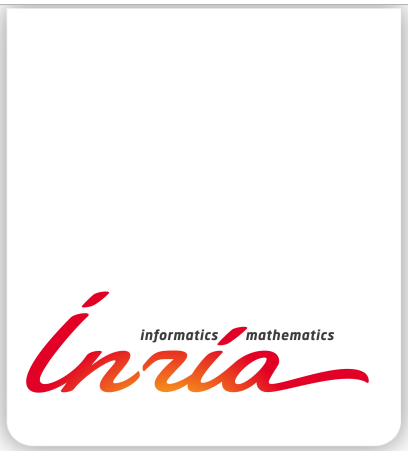
$\langle 1 \rangle 6$. CASE: W_{tt} and W_{ff}

$\langle 2 \rangle 1$. Q.E.D.

Same as conditionals.

$\langle 1 \rangle 7$. Q.E.D.

PROOF: By induction on instructions evaluation and $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399