Block-safe Information Flow Control

Elisavet Kozyri*
Cornell University
ekozyri@cs.cornell.edu

Josée Desharnais†
Laval University
josee.desharnais@ift.ulaval.ca

Nadia Tawbi†
Laval University
tawbi@ulaval.ca

August 8, 2016

Abstract

Flow-sensitive dynamic enforcement mechanisms for information flow labels offer increased permissiveness. However, these mechanisms may leak sensitive information when deciding to block insecure executions. When enforcing two labels (e.g., *secret* and *public*), sensitive information is leaked from the context in which this decision is taken. When enforcing arbitrary labels, additional sensitive information is leaked from the labels involved in the decision to block an execution. We give examples where, contrary to a common belief, a mechanism designed to enforce two labels may not be able to enforce arbitrary labels, due to this additional leakage. In fact, it is not trivial to design a dynamic enforcement that offers increased permissiveness, handles multiple labels, and does not introduce information leakage due to blocking insecure executions. In this paper, we present a dynamic enforcement mechanism of information flow labels that has all these three attributes. Our mechanism is not *purely dynamic*, since it uses a light-weight, on-the-fly, static analysis of untaken branches. We prove that the set of all normally terminated and blocked traces of a program, which is executed under our mechanism, satisfies noninterference, against principals that make observations throughout execution.

*Supported in part by AFOSR grants F9550-16-0250 and grants from Microsoft.
†Research supported by NSERC.

1 Introduction

Flow-sensitive dynamic (FSD) enforcement of information flow policies has been the subject of an extended line of research (e.g., [12, 9, 5, 4, 13, 28, 21]), because it offers increased permissiveness (i.e., it rejects fewer secure executions) and programming flexibility (e.g., it can handle dynamically loaded code). FSD mechanisms use run time information (i) to automatically deduce information flow labels for computed values, and (ii) to guarantee the satisfaction of these labels. For confidentiality, an information flow label that tags a value is satisfied if the value and all its derived values are read only by principals specified by the label.

The majority of FSD mechanisms (i.e., [25, 12, 1, 7, 9, 5, 4, 29, 13, 28, 21, 23, 8]) monitor program executions and take some action when the next command to be executed may violate labels on some values. These mechanisms can either modify this insecure command, skip it, execute an infinite loop, or block the execution ([24, 23, 12]). The most common action, which is also employed by work on FSD mechanisms for JavaScript [20, 13, 28, 21], is blocking the execution. In this paper, we focus on FSD mechanisms that block an execution when the remaining command to be executed may violate labels on some values.

An FSD mechanism that blocks insecure executions may introduce covert channels that do not already exist in programs [10]. Indeed, such a covert channel may be introduced if the decision to block an execution depends on sensitive information. We call mechanisms that introduce covert channels by blocking executions, *block-unsafe*; mechanisms that do not introduce such channels are called *block-safe*.

A block-unsafe mechanism may leak entire secrets, if principals make observations during program execution [2].[1] For example, consider the following program:

$$
\begin{array}{l}
w := 0; \\
\textbf{while } (w \leq N)\{ \\
\quad \textbf{if } (w = h) \textbf{ then } l' := h \textbf{ end} \\
\quad w := w + 1 \\
\quad l := w \\
\}
\end{array}
\tag{1}
$$

[1] It may leak one bit per execution if principals observe only the final values of variables, at the end of execution. In this paper we consider the strongest threat model where principals make observations throughout the execution.

where h is sensitive information, l, l' store public information that all principals can observe, and w's sensitivity changes to match the sensitivity of the information stored in w. The only insecure command in the above program is assignment $l' := h$, because it makes a sensitive information (i.e., h) observable to all principals (i.e., those observing l'). A block-unsafe FSD mechanism would block the execution only if this insecure assignment $l' := h$ is the next command to be executed. However, when the execution is blocked, l contains the exact value of h (assuming $0 \leq h \leq N$), and thus, the sensitive value of h is leaked to all principals, which observe l.

Previously proposed block-safe mechanisms [7, 1][2] are not as permissive as block-unsafe mechanisms (e.g., [20, 13, 28, 21]). This is due to the way labels on variables are constructed after the termination of *conditional commands*.[3] Consider, for example, the following conditional command:

$$\textbf{if } l > 0 \textbf{ then } w := h \textbf{ else } w := l \textbf{ end}. \tag{2}$$

Previously proposed block-safe mechanisms always associate label `H` (*high* confidentiality) with w at the end of this conditional command, whereas block-unsafe mechanisms associate `H` with w, when $l > 0$, and associate `L` (*low* confidentiality) with w, when $l \leq 0$. So, the former mechanisms make a conservative approximation when updating w's label, while the latter consider the actual label of the value stored in w. In this paper, we call a FSD mechanism *true flow-sensitive*, if at the end of conditional commands, it considers the actual labels of values stored in variables. True flow-sensitive mechanisms offer increased permissiveness, because they tag variables with less conservative labels.

The goal for this paper is to design a dynamic mechanism for multiple labels that is both block-safe and true flow-sensitive. It is tricky to design such a mechanism. Take, for example, the true flow-sensitive mechanism used in example (1). In this example, sensitive information h is leaked because the mechanism decided to block the execution in a context that depends on h. Consider, instead, a true flow-sensitive mechanism that is checking the security of assignments before entering a sensitive context. In this way, the decision to block an execution would not depend on sensitive information. Such a mechanism is block-safe when only two labels are used.[4]

[2] They are *termination sensitive*, which is stronger than block-safe.

[3] A conditional command is either an "if" or "while" command.

[4] Checks before conditional commands are used in [3], but for a flow-insensitive mechanism. We believe that the true flow-sensitive version of this mechanism is still block-safe for two labels.

Surprisingly, it is not block-safe when multiple labels are used. Take, for example, the following program that uses three labels:

$$\begin{array}{l} l := 0; \\ \textbf{if } m > 0 \textbf{ then } w := m \textbf{ else } w := h \textbf{ end} \\ m' := w; \\ l := 1 \end{array} \tag{3}$$

where l is tagged with `L`, h is tagged with `H`, and m, m' are tagged with *medium* label `M` (i.e., medium confidentiality). Even if no check on labels is performed in sensitive context, the sensitive value stored in m could still be leaked to l via blocking executions, as we now explain. Assignment $m' := w$ is allowed to be executed when the label of w is not violated by the label of m' (i.e., `M`), and it is blocked, otherwise. Notice that this check on labels is not performed in a sensitive context. If $m > 0$, then the label of w is `M`, assignment $m' := w$ is not blocked, and thus, l becomes 1. If $m \not> 0$, then the label of w is `H`, assignment $m' := w$ is blocked, assignment $l := 1$ is not executed, and thus the value of l remains 0. Thus, deciding to block assignment $m' := w$ causes sensitive value $m > 0$ to be leaked to variable l. This example challenges a common belief that mechanisms for two labels can be easily extended to offer the same security guarantees for arbitrary labels.

In this paper, we introduce a block-safe, true flow-sensitive, dynamic mechanism that enforces multiple labels. To provide block-safety we keep track of the sensitivity of the information used to decide whether an execution blocks, and we ensure that this information is not leaked to unauthorized principals. The decision to block an execution depends on the context in which this decision is taken and on the labels of variables that are being checked. These labels may depend on sensitive information, due to the combination of true flow-sensitivity and the use of more than two labels (in Example (3), the label of w depends on m). We employ *metalabels*, which are labels on labels, to represent the sensitivity of corresponding labels. The sensitivity of a check is then determined by the metalabels of labels involved in this check, as well as, the sensitivity of the context in which this check is performed. An execution is blocked when the sensitivity of previously evaluated checks may be violated.

Metalabels are computed during execution, and thus, they may as well depend on sensitive information. Fortunately, we do not need additional machinery to protect sensitive information encoded in metalabels (e.g., we do not need meta-metalabels) from leaking to unauthorized principals via

blocking executions. The way our mechanism uses metalabels is sufficient to guarantee *noninterference* [18].

We prove that the set of all normally terminated and blocked traces of a program, which is executed under our mechanism, satisfies noninterference. So, we prove that changes in initial sensitive information do not influence the observations made by unauthorized principals throughout execution, no matter if the executions were blocked or terminated normally. Thus, our mechanism is block-safe. To address implicit flow of information that could violate noninterference, our mechanism employs a light-weight, on-the-fly, static analysis of untaken branches.

From our mechanism we derive a simpler one that handles two labels (`H` and `L`). Due to true flow-sensitivity, this simpler mechanism is, in many cases, more permissive than other block-safe FSD mechanisms already proposed for two labels.

We proceed as follows: Section 2 gives the programming language in which programs are assumed to be written, and it defines the threat model. Section 3 presents our enforcement mechanism, and Section 4 gives the soundness theorem for our mechanism. Section 5 discusses the derivation of a simpler mechanism for `H` and `L` labels. Section 6 examines related work, and Section 7 concludes.

2 Labels and Programming Language

Our mechanism tags each variable with a label ℓ from a set $\mathcal{L}$. Values stored in a variable tagged with ℓ, are considered to be tagged with ℓ, too. Thus, a label ℓ that tags a variable x, dictates who can read a value v stored in x and all values that may be derived from v. We say that label ℓ is *at least as restrictive as* label ℓ', writing $\ell' \sqsubseteq \ell$, if ℓ allows a subset of principals to read values than those allowed by ℓ'. So, a value tagged with ℓ is considered to be at least as sensitive as a value tagged with ℓ'.

In security literature, the pair $\langle \mathcal{L}, \sqsubseteq \rangle$ is a lattice, with bottom element $\bot$ and join operator $\sqcup$. Bottom element $\bot$ represents the least restrictive policy. The result of applying $\sqcup$ to two labels ℓ_1, ℓ_2 is a third label ℓ_3 that represents a policy at least as restrictive as the policy represented by ℓ_1 and ℓ_2. Label ℓ_3 is the least restrictive label that satisfies this description. A lattice commonly used in information flow literature is $\langle \{\mathtt{H}, \mathtt{L}\}, \sqsubseteq \rangle$, where $\bot = \mathtt{L}$ and $\mathtt{L} \sqsubseteq \mathtt{H}$. Here, $\mathtt{L} \sqcup \mathtt{H} = \mathtt{H}$, $\mathtt{H} \sqcup \mathtt{H} = \mathtt{H}$, and $\mathtt{L} \sqcup \mathtt{L} = \mathtt{L}$. If a variable x is tagged with `L`, then all principals are allowed to read x (x is public), whereas if x is tagged with `H`, then only a particular set of principals are

allowed to read x (x is secret).

For this paper, we consider programs being written in a simple imperative programming language, whose syntax is presented in Figure 1. This language defines two disjoint sets of variables:

Anchor variables $\mathcal{V}_a$**:** Variables with *fixed* labels (i.e., their labels do not change during execution). Anchor variables simulate sources and sinks of information with fixed labels, such as files, channels and input/output devices. An anchor variable is denoted with a. We also use h, m, and l to denote anchor variables tagged with labels `H`, `M`, and `L`, correspondingly.

Working variables $\mathcal{V}_w$**:** Variables with changing labels. Working variables simulate temporary slots used during computation, such as memory slots and registers. Labels for working variables are deduced by our dynamic mechanism. We use w to denote a working variable, and x to denote a working or an anchor variable.

So, $\mathcal{V} = \mathcal{V}_a \cup \mathcal{V}_w$ is the complete set of variables.

$$
\begin{array}{lrcl}
\textit{(Constants)} & n & \in & N \\
\textit{(Anchor variables)} & a, x & \in & \mathcal{V}_a \\
\textit{(Working variables)} & w, x & \in & \mathcal{V}_w \\
\textit{(Expressions)} & \mathcal{E} & ::= & n \mid x \mid \mathcal{E}_1 \ \mathbf{op}\ \mathcal{E}_2 \\
\textit{(Commands)} & \mathcal{C} & ::= & \mathbf{skip} \mid x := \mathcal{E} \mid \mathcal{C}_1; \mathcal{C}_2 \mid \\
 & & & \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2\ \mathbf{end} \mid \\
 & & & \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end}
\end{array}
$$

Figure 1: Syntax

According to the syntax presented in Figure 1, an expression $\mathcal{E}$ is either a constant, or an anchor variable, or a working variable, or a synthesis of expressions. A command is either **skip**, or an assignment, or a conditional command, or a synthesis of commands.

The execution of a program is semantically represented by a *trace* Σ, which is a sequence of program *states*.[5] A program state σ is a pair $\langle \mathcal{C}^-,\ M \rangle$ consisting of a *residual command* $\mathcal{C}^-$ and *memory* M. A residual command $\mathcal{C}^-$ is a remaining command to be executed, defined as:

$$\mathcal{C}^- \triangleq \mathcal{C} \mid \mathbf{stop} \mid \mathbf{end} \mid \mathcal{C}_1^-; \mathcal{C}_2^-.$$

[5]Section 3 gives the operational semantics that generates these traces.

Residual command **stop** is an indication of normal termination, and **end** informs the enforcement mechanism that a conditional command has been exited. The information that a conditional command has been exited is used by our enforcement mechanism to properly update labels of certain variables.

A memory M maps variables to values and labels. $M(x)$ denotes the value of variable x in M, and $M(\underline{x})$ denotes the label of anchor or working variable x in M.

Consequently, a trace Σ is represented by a sequence

$$\langle \mathcal{C}_1^-,\ M_1\rangle \rightarrow \langle \mathcal{C}_2^-,\ M_2\rangle \rightarrow \ldots \rightarrow \langle \mathcal{C}_n^-,\ M_n\rangle.$$

We write $\langle \mathcal{C}_1^-,\ M_1\rangle \stackrel{*}{\rightarrow} M_n$ to denote a trace that starts from state $\langle \mathcal{C}_1^-,\ M_1\rangle$ and normally terminates at a state $\langle \textbf{stop},\ M_n\rangle$, or it is blocked, due to the enforcement mechanism, at a state $\langle \mathcal{C}_n^-,\ M_n\rangle$.

Threat Model

During execution, principals can observe values being assigned to variables. Each principal is associated with a label that indicates an upper bound on the sensitivity of values that this principal can observe. Our mechanism uses predicate $\phi(M, x, \ell)$ to encode whether the assignment of a value to variable x can be observed by a principal associated with ℓ. Predicate ϕ is formally defined in the next section. Additionally, all principals can sense the (normal or enforced) termination of an execution. After termination, no more observations are generated.

So, a trace Σ can be projected onto a *sequence of observations* $\Sigma|_\ell$ that is made by a principal associated with ℓ. Specifically, $\Sigma|_\ell$ evaluates to a list of pairs $\langle x,\ M(x)\rangle$, where there is one pair for each assignment $x := \mathcal{E}$ to a *target* variable x satisfying $\phi(M, x, \ell)$ that occurred in trace Σ.

Multiple principals can make observations from a trace. If L is a set of labels, then a trace Σ can be projected onto a sequence of observations $\Sigma|_L$ that is made by principals associated with some labels in L. Formally, given a trace $\Sigma = \langle \mathcal{C}_1^-,\ M_1\rangle \rightarrow \langle \mathcal{C}_2^-,\ M_2\rangle \rightarrow \Sigma'$, where Σ' may be empty, we

define:

$$(\langle \mathcal{C}_1^-,\ M_1\rangle \rightarrow \langle \mathcal{C}_2^-,\ M_2\rangle \rightarrow \Sigma')|_L \triangleq$$
$$\begin{cases} \langle x,\ M_2(x)\rangle \rightarrow & \text{if } \mathcal{C}_1^- \text{ is "} x := \mathcal{E}; \mathcal{C}^- \text{"} \\ (\langle \mathcal{C}_2^-,\ M_2\rangle \rightarrow \Sigma')|_L & \text{and } \exists \ell \in L.\phi(M_2, x, \ell) \\ \\ (\langle \mathcal{C}_2^-,\ M_2\rangle \rightarrow \Sigma')|_L & \text{otherwise.} \end{cases}$$

If Σ is empty, or if Σ consists of only one state $\langle \mathbf{stop},\ M\rangle$, then $\Sigma|_\ell$ is empty.[6]

Notice that, according to our threat model, assignments to both anchor and working variables may generate observations to principals. Because an anchor variable is tagged with a fixed label, always the same principals are allowed to observe assigned values to that anchor variable. For a working variable, however, different sets of principals are allowed to observe assigned values during execution, because different labels may tag this variable during execution.

Our enforcement mechanism ensures that sequences of observations do not leak sensitive information to unauthorized principals.

3 Enforcement mechanism

We present a block-safe, true flow-sensitive mechanism that dynamically enforces labels on variables, (i) by *redefining* labels for working variables, given labels for anchor variables, and (ii) by blocking executions that may subsequently violate these labels. True flow-sensitivity allows increased precision in how labels are redefined. Block-safety prevents information leakage when blocking insecure executions.

3.1 True flow-sensitivity

First we explain how our mechanism provides true flow-sensitivity. Labels of working variables are redefined during execution depending on the sensitivity of values that are stored in them (aka flow-sensitivity). A label of a working variable must account for explicit and implicit flows of information. An explicit flow is caused by the execution of an assignment where the value

[6] One could have modeled the fact that all principals sense termination, by adding one special observation at the end of all sequences of observations. However this special observation is not necessary to prove the desired soundness theorem.

$$\text{(S-Skip)}\ \frac{}{\langle \mathbf{skip},\ M\rangle \to \langle \mathbf{stop},\ M\rangle}$$

$$\text{(S-Asgn1)}\ \frac{\begin{array}{c} v = M(\mathcal{E}) \qquad M(\underline{\mathcal{E}}) \sqcup M(pc).label \sqcup block' \sqsubseteq M(\underline{a}) \\ block' = M(block) \sqcup M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label \end{array}}{\langle a := \mathcal{E},\ M\rangle \to \langle \mathbf{stop},\ M[a \mapsto v, block \mapsto block']\rangle}$$

$$\text{(S-Asgn2)}\ \frac{v = M(\mathcal{E}) \qquad \ell = M(\underline{\mathcal{E}}) \sqcup M(pc).label \qquad \ell' = M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label}{\langle w := \mathcal{E},\ M\rangle \to \langle \mathbf{stop},\ M[w \mapsto v, \underline{w} \mapsto \ell, \underline{\underline{w}} \mapsto \ell']\rangle}$$

$$\text{(S-Seq1)}\ \frac{\langle \mathcal{C}_1^-,\ M\rangle \to \langle \mathbf{stop},\ M'\rangle}{\langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \to \langle \mathcal{C}_2^-,\ M'\rangle}$$

$$\text{(S-Seq2)}\ \frac{\langle \mathcal{C}_1^-,\ M\rangle \to \langle \mathcal{C}_{1'}^-,\ M'\rangle \qquad \mathcal{C}_{1'}^- \neq \mathbf{stop}}{\langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \to \langle \mathcal{C}_{1'}^-; \mathcal{C}_2^-,\ M'\rangle}$$

$$\text{(S-If1)}\ \frac{\begin{array}{c} M(\mathcal{E}) \neq 0 \qquad V = targetWVar(\mathcal{C}_2) \\ d = anchorVar(\mathcal{C}_2) \qquad pc' = M(pc).push(\langle M(\underline{\mathcal{E}}),\ d,\ V\rangle) \end{array}}{\langle \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2\ \mathbf{end},\ M\rangle \to \langle \mathcal{C}_1; \mathbf{end},\ M[pc \mapsto pc']\rangle}$$

$$\text{(S-If2)}\ \frac{\begin{array}{c} M(\mathcal{E}) = 0 \qquad V = targetWVar(\mathcal{C}_1) \\ d = anchorVar(\mathcal{C}_1) \qquad pc' = M(pc).push(\langle M(\underline{\mathcal{E}}),\ d,\ V\rangle) \end{array}}{\langle \mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{C}_1\ \mathbf{else}\ \mathcal{C}_2\ \mathbf{end},\ M\rangle \to \langle \mathcal{C}_2; \mathbf{end},\ M[pc \mapsto pc']\rangle}$$

$$\text{(S-Wl1)}\ \frac{M(\mathcal{E}) \neq 0 \qquad pc' = M(pc).push(\langle M(\underline{\mathcal{E}}),\ false,\ \emptyset\rangle)}{\langle \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M\rangle \to \langle \mathcal{C}; \mathbf{end}; \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M[pc \mapsto pc']\rangle}$$

$$\text{(S-Wl2)}\ \frac{\begin{array}{c} M(\mathcal{E}) = 0 \qquad V = targetWVar(\mathcal{C}) \\ d = anchorVar(\mathcal{C}) \qquad pc' = M(pc).push(\langle M(\underline{\mathcal{E}}),\ d,\ V\rangle) \end{array}}{\langle \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M\rangle \to \langle \mathbf{end},\ M[pc \mapsto pc']\rangle}$$

$$\text{(S-End)}\ \frac{\begin{array}{c} \text{IF } M(pc).top.d \text{ THEN } block' = M(block) \sqcup M(pc).label \text{ ELSE } block' = M(block) \\ M' = U(M, M(pc).top.V) \qquad pc' = M(pc).pop \end{array}}{\langle \mathbf{end},\ M\rangle \to \langle \mathbf{stop},\ M'[pc \mapsto pc', block \mapsto block']\rangle}$$

Table 1: Enforcement mechanism

of the right hand-side expression explicitly flows to the target variable. An implicit flow is caused by the execution of a conditional command, where the value of the guard implicitly flows to the target variable of each assignment in the branches. We further examine how our mechanism redefines labels for working variables based on these two kinds of flows.

When assignment $w := \mathcal{E}$ is executed, the value of expression $\mathcal{E}$ explicitly flows to w. So, the new value of w should be regarded at least as sensitive as the value of $\mathcal{E}$, and thus, the new label of w should be at least as restrictive as the label of $\mathcal{E}$. We denote with $M(\underline{\mathcal{E}})$ the label of an expression $\mathcal{E}$, according to memory M. $M(\underline{\mathcal{E}})$ is defined inductively:

- $M(\underline{n}) \triangleq \bot$,
- $M(\underline{x})$ is by definition the label associated with x,
- $M(\underline{\mathcal{E}_1 \,\mathbf{op}\, \mathcal{E}_2}) \triangleq M(\underline{\mathcal{E}_1}) \sqcup M(\underline{\mathcal{E}_2})$.

The execution of a conditional command causes the value of its guard to implicitly flow to target variables in the branches. So, the labels of target variables in all branches (both taken and untaken branches) should be properly redefined to account for the sensitivity of the guard. Specifically, target working variables should become at least as sensitive as the value of the guard. A *program counter stack pc* keeps track of the sensitivity of the context of a command, which is the sensitivity of all guards that control whether this command is executed. Whenever an execution enters a conditional command, the label of its guard is pushed into $M(pc)$, and whenever an execution leaves a conditional command, the top element of $M(pc)$ is popped. The join of all labels in $M(pc)$, which is denoted with $M(pc).\mathit{label}$, is the sensitivity of the information used for the execution to reach the current program point. If $M(pc)$ is empty, then $M(pc).\mathit{label} = \bot$. So, the label of a target working variable should be at least as restrictive as $M(pc).\mathit{label}$.

The strategy we follow to redefine the labels of target working variables depends on whether the corresponding assignment is actually executed (e.g., belongs to the taken branch), or could have been executed (e.g., belongs to the untaken branch). When assignment $w := \mathcal{E}$ is executed in memory M, the dynamic mechanism tags w with label $M(\underline{\mathcal{E}}) \sqcup M(pc).\mathit{label}$, because the value of w possibly reveals information about $\mathcal{E}$ and about the context in which the assignment is executed (see the update of $\underline{w}$ in (S-Asgn2), Table 1).

Labels of target working variables that belong to the untaken branch of an executed conditional command are *silently* redefined. When the execution of a conditional command ends, the labels of these working variables are augmented with the labels of all guards that control the execution of the corresponding assignments, or else, the labels of these working variables are augmented with $M(pc).label$. To accomplish this, before entering a conditional command, function *targetWVar* is used to collect all target working variables of the untaken branch.[7] These variables V are pushed onto $M(pc)$, together with the label of the corresponding guard. When finishing executing a conditional command, and before popping the top element of $M(pc)$, the dynamic mechanism augments the labels of all variable in V with $M(pc).label$, using function U.[8] Notice that function *targetWVar* implements a light-weight, on-the-fly static analysis on the untaken branches of executed conditional commands (see use of V in (S-IF1), (S-IF2), (S-WL2), and use of U in (S-END), Table 1).

The way we update labels of working variables at the end of a conditional command is what gives true flow-sensitivity. Past work (e.g., [9, 5, 19, 13]) use similar techniques to provide true flow-sensitivity. Other approaches, which are not true flow-sensitive (e.g, [7, 1]), use static analysis (either on-the-fly or as a prepossessing stage) to augment labels of working variables with labels that these variables would have taken if the untaken branch had been executed, instead. This augmentation involves both $M(pc).label$ and labels of right hand-side expressions of assignments. So, a mechanism that is not true flow-sensitive tags a working variable w with $\ell_1 \sqcup \ell_2$, at the end of a conditional command, where ℓ_1 is the redefined label of w due to taken branch, and ℓ_2 is the redefined label of w due to untaken branch. Because we augment these working variables only with $M(pc).label$, the redefined labels are less conservative, leading to a more permissive mechanism. Considering example (2), not true flow-sensitive approaches always tag w with `H` at the end of the conditional command, whereas true flow-sensitive approaches tag w with either `H` or `L` depending on the branch that is taken.

Static analysis of untaken branches is not the only way to provide true flow-sensitivity. Purely dynamic mechanisms (e.g., [4, 5]), which do not involve static analysis, can provide true flow-sensitivity, too. However, previously proposed purely dynamic mechanisms (e.g., [13, 28, 21]) are block-unsafe, because they may leak sensitive information when deciding to block insecure executions.

[7] $targetWVar(\mathcal{C}) \triangleq \{w \mid \text{“}w := \mathcal{E}\text{”} \in \mathcal{C}\}$

[8] U is formally defined later in this section.

3.2 Block-safety

Now, we turn our attention to how our mechanism provides block-safety. Assignments to anchor variables may cause violation of labels on variables, when the fixed label of a target anchor variable is not at least as restrictive as the label of the value that flows to that anchor variable. Our mechanism blocks program execution before such insecure assignments. To provide block-safety, we need to ensure that sensitive information used to check the security of an assignment does not leak to unauthorized principals.

Indeed, an assignment check may depend on sensitive information. First, this check is performed at a particular program point, before a particular assignment. Also, this check involves the label of the target anchor variable a of the assignment and labels of working and anchor variables whose value flow to a. So, an assignment check may reveal sensitive information used:

1. to reach that program point, and
2. to redefine labels of these working variables.

As an example of point 1, consider the following command:

$$\textbf{if } h > 0 \textbf{ then } l := h' \textbf{ else } h' := l \textbf{ end}; l := 1.$$

Here, checking assignment $l := h'$ depends on the sensitive information $h > 0$. If the mechanism is not careful, performing the check and deciding to block that assignment may leak sensitive information $h > 0$ to l. This is because, when $h \not> 0$, no check is performed, the execution terminates normally, and thus, some principals observe 1 being assigned to l. However, when $h > 0$, the check is performed, the execution is blocked, and thus, no assignment to l is observed by these principals.

As an example of point 2, consider program (3). Checking the security of assignment $m' := w$ depends on the label of w, which in turn depends on the sensitive information $m > 0$. Again, as described in Section 1, if the mechanism is not careful, checking the security of $m' := w$ and deciding to block that assignment may leak sensitive information $m > 0$ to l.

Our block-safe mechanism guarantees that the sensitive information used for assignment checks does not leak to unauthorized principals. The strategy we follow is two-fold:

(i) Track the sensitivity of each check, by tracking the sensitivity of the context in which this check is performed (to address point 1) and the sensitivity of redefined labels used in this check (to address point 2),

(ii) Ensure that, at every execution point, the observations made by principals do not violate the sensitivity of any check that precedes this execution point.

For point (i), *pc* tracks the sensitivity of the context, and metalabels track the sensitivity of redefined labels. For each variable x, we introduce a metalabel $\underline{\underline{x}}$ that represents the sensitivity of the information used to redefine label $\underline{x}$ of x. In particular, metalabel $\underline{\underline{x}}$ specifies the principals (i.e., those associated with a label at least as restrictive as $\underline{x}$) that can learn $\underline{x}$ without violating labels of input values (i.e., labels of anchor variables). So, $M(\underline{x})$ is a label ($M(\underline{x}) \in \mathcal{L}$) that protects $M(\underline{x})$, the same way $M(\underline{x})$ protects $M(x)$. Metalabels of anchor variables are by default the bottom label $\bot$, because their labels never change, and so they cannot encode sensitive information. Metalabels of working variables are redefined whenever labels for these variables are redefined, which happens whenever new values are assigned to these working variables.

We explain how metalabels of working variables are redefined. We saw that when assignment $w := \mathcal{E}$ is executed in memory M, label $\underline{w}$ of w becomes $M(\underline{\mathcal{E}}) \sqcup M(pc).label$. Redefining the label of w is based on the label of $\mathcal{E}$ and on the context in which this assignment is executed. So, the redefined label of w is at least as sensitive as the label of $\mathcal{E}$ and the context. Thus, the metalabel of w should be at least as restrictive as the metalabel of $\mathcal{E}$ and $M(pc).label$. So, the metalabel of w becomes $\underline{\underline{\mathcal{E}}} \sqcup M(pc).label$ (see update of $\underline{\underline{w}}$ in (S-Asgn2)), where $\underline{\underline{\mathcal{E}}}$ is defined inductively:

- $M(\underline{\underline{n}}) \triangleq \bot$,
- $M(\underline{\underline{a}}) \triangleq \bot$, $M(\underline{\underline{w}})$ is by definition the metalabel associated with w,
- $M(\underline{\underline{\mathcal{E}_1 \,\mathbf{op}\, \mathcal{E}_2}}) \triangleq M(\underline{\underline{\mathcal{E}_1}}) \sqcup M(\underline{\underline{\mathcal{E}_2}})$.

Notice that, when assignment $w := \mathcal{E}$ is executed, $\underline{\underline{\mathcal{E}}}$ accounts for the information that explicitly flows from $\underline{\mathcal{E}}$ to $\underline{w}$, while $M(pc).label$ accounts for the information that implicitly flows from the context to $\underline{w}$.

To capture all implicit flows of information from the context to labels, our mechanism silently redefines the metalabels of target working variables found in untaken branches. Consider the following example:

$$w := l;\ \mathbf{if}\ m > 0\ \mathbf{then}\ w := h\ \mathbf{else\ skip\ end}.$$

Sensitive information m implicitly flows to $\underline{w}$. So, $\underline{w}$ should be at least as sensitive as m, and thus $\underline{\underline{w}}$ should be redefined to at least `M` by the end of all

executions of this command. If $m > 0$, and assignment $w := h$ is executed, then, as we saw in the previous paragraph, $\underline{\underline{m}}$ becomes $\mathtt{H} \sqcup \mathtt{M}$, which is at least as restrictive as $\mathtt{M}$. If $m \not> 0$, then our mechanism silently redefines $\underline{\underline{w}}$ to $\underline{\underline{w}} \sqcup \mathtt{M}$. Using function U[9], both metalabels and labels of target working variables in untaken branches are augmented with the sensitivity $M(pc).label$ of the context.

We can now formally characterize the sensitivity of an assignment check (i.e., point (i)). When assignment $a := \mathcal{E}$ is next to be executed, it should be checked whether the label of a is more restrictive than the label of $\mathcal{E}$ (to prevent explicit flows) and more restrictive that $M(pc).label$ (to prevent implicit flows). The evaluation of this check depends on the label $\underline{\mathcal{E}}$ of $\mathcal{E}$, and on the context of the assignment. So, principals allowed to learn the result of this check are those allowed to learn both the value of $\underline{\mathcal{E}}$ and information about the context. Thus, the evaluation of this check is at least as sensitive as $\underline{\mathcal{E}}$ and the context. Thus, the sensitivity of this assignment check is $\underline{\underline{\mathcal{E}}} \sqcup M(pc).label$.

For point (ii), we use a variable *block* [7, 1] to ensure that the sensitivity of checks is not violated by subsequent observations. Specifically, whenever a check is performed, *block* is augmented with the sensitivity of this check, which is the join of the metalabels of all labels involved in this check and the label of the context in which this check is performed (see update of *block* in (S-Agn1)). So, at every execution point, *block* represents the sensitivity of all checks that determine whether the execution will reach that point or be blocked earlier.

Variable *block* is used by our mechanism to decide which observations will be generated for which principals. According to Section 2, observations are generated when values are assigned to anchor or working variables. When assignment $a := \mathcal{E}$ to an anchor variable a is executed, the fixed label that tags a dictates principals that can observe values being assigned to a. The assignment of a value to an anchor variable is allowed to be observed, only if the label of this anchor variable is more restrictive than *block*. Otherwise, information used to evaluate previous checks would be leaked to less sensitive anchor variable, and thus, to principals not authorized to learn anything about this information. Thus, if the label of a target anchor variable is not at least as restrictive as *block*, the execution blocks before executing the corresponding assignment. So, now, each decision to block an execution also involves *block* (see (S-Asgn1)).[10]

[9] Function U is defined as follows:
$U(M, V) \triangleq M[\forall x \in V : M(\underline{x}) = M(\underline{x}) \sqcup M(pc).label, M(\underline{\underline{x}}) = M(\underline{\underline{x}}) \sqcup M(pc).label]$

[10] For clarity, we write both $M(pc).label$ and $block'$ in restriction $M(\underline{\mathcal{E}}) \sqcup M(pc).label \sqcup$

Observations generated by assigning values to working variables are restricted by variable *block*, too. The assignment of a new value to a working variable w is observed only by principals whose label is at least as restrictive as the redefined label of w, and variable *block*. Notice that an assignment to a working variable is always executed, but our mechanism restricts the principals that make the corresponding observation.

In general, a principal is allowed to observe a value being assigned to an anchor or working variable x, only if her label is more restrictive than both $M(\underline{x})$ and $M(block)$. Thus, predicate $\phi(M, x, \ell)$, which was presented in §2 and which defines observations allowed to be made by principals associated with ℓ, is defined to be:

$$\phi(M, x, \ell) \triangleq M(\underline{x}) \sqcup M(block) \sqsubseteq \ell. \tag{4}$$

This concludes point (ii) of our strategy.

Sensitive information used to compute *block* should not leak to unauthorized principals. Variable *block* may be updated in a conditional command with a sensitive guard. So, the value of *block* may be different at the end of this conditional command, depending on the branch that is taken. To capture implicit flows from a sensitive context to variable *block*, our mechanism silently redefines *block* at the end of conditional commands, by augmenting *block* with $M(pc).label$. When the execution enters a conditional command, we use function *anchorVar* to find whether there is an assignment to anchor variables in the untaken branch, and thus, whenever *block* could have been updated in the untaken branch.[11] We push onto *pc* this truth value d. When exiting a conditional command, if the top element of *pc* has $d = true$, then *block* is updated with $M(pc).label$. So, *pc* is actually a stack of triples, where each triple contains a label, a truth value d, and a set V of working variables.[12] Notice that function *anchorVar* implements a light-weight, on-the-fly, static analysis of the untaken branches (see use of d in (S-If1), (S-If2), (S-Wl2), and update of *block* in (S-End)).

Table 1 contains the operational semantics for executing a program under our true flow-sensitive and block-safe mechanism.

$block' \sqsubseteq M(\underline{a})$, even though we have $M(pc).label \sqsubseteq block'$, which implies $M(pc).label \sqcup block' = block'$.

[11] $anchorVar(\mathcal{C}) \triangleq$ Is there any "$a := \mathcal{E}$" in $\mathcal{C}$?

[12] $M(pc).label$ is formally defined as:
$M(pc).label \triangleq \bigsqcup_{\langle \ell,\ d,\ V\rangle \in pc} \ell$.
$M(pc).push(\langle \ell,\ d,\ V\rangle)$ pushes element $\langle \ell,\ d,\ V\rangle$ on the top of stack $M(pc)$ and returns the resulting stack.
$M(pc).pop$ pops the top element of stack $M(pc)$ and returns the resulting stack.
If $M(pc).top$ is $\langle \ell,\ d,\ V\rangle$, then $M(pc).top.d$ is d and $M(pc).top.V$ is V.

3.3 Examples

Now, we discuss how our mechanism handles particular small examples. First, consider example (1). Our mechanism enters this loop only once. If $w = h$, then using rule (S-Asgn1) the execution is blocked at assignment $l' := h$, because anchor variable l' is not at least as restrictive as h. If $w \neq h$, then due to (S-If2) and (S-End), *block* becomes `H` after the **if** command. Then, assignment $w := w + 1$ is executed according to rule (S-Asgn2). But assignment $l := w$ is blocked due to (S-Asgn1) and the fact that label `L` of l is not more restrictive than label `H` in variable *block*. In both executions, no observation is generated to principals associated with `L`, and thus, h does not leak to these principals. In particular, blocking the execution of this program in different points, depending on h, does not leak h.

Next, consider example (3). If $m > 0$, then using (S-Asgn2), $\underline{w}$ and $\underline{\underline{w}}$ become `M`. So, due to (S-Asgn1), assignment $m' := w$ is allowed to be executed, while *block* is being updated to $\underline{\underline{w}}$, which is `M`. Next assignment $l := 1$ is not executed, due to (S-Asgn1), where label `L` of l is not at least as restrictive as label `M` in variable *block*. If $m \not> 0$, then using (S-Asgn2), $\underline{w}$ becomes `H` and $\underline{\underline{w}}$ becomes `M`. Next assignment $m' := w$ is blocked due to (S-Asgn1), where label `M` of m' is not at least as restrictive as label `H` of w. Thus, blocking the execution of this program in different points, depending on the label of w, which depends on sensitive information m, did not leak m to principals associated with `L`.

Finally, consider the following program:

$$\begin{array}{l}\textbf{if } m > 0 \textbf{ then } w := h \textbf{ else } w := h' \textbf{ end};\\ h := w;\\ l := 1.\end{array}$$

This program is secure. However, our mechanism would block all executions of this program before assignment $l := 1$. This is because, $\underline{\underline{w}}$ becomes `M`, at the end of the conditional command, *block* becomes `M` after executing $h := w$, and the label `L` of target anchor variable l is not at least as restrictive as *block*. So, our mechanism conservatively deduces that the label $\underline{w}$ of w depends on m, even though, in reality, there is no such dependency ($\underline{w}$ becomes `H` in both branches).

4 Soundness

We prove soundness of our mechanism by showing that programs executed under our mechanism (Table 1) satisfy noninterference for an arbitrary lat-

tice of labels. We prove that commands allowed to be executed and actions taken by our mechanism do not leak sensitive information to observations made by unauthorized principals. The actions taken by our mechanism that may affect observations are (i) redefinition of labels for working variables, and (ii) blocking of insecure executions. So, we prove that redefining labels and deciding to block executions do not leak information. We are not concerned with information leakage introduced by divergence of while-loops that exist in a program. Thus, we prove noninterference for the set of finite-length traces that either terminated normally, or were blocked.

Previous work on true flow-sensitive mechanisms [9, 4, 12, 13, 28, 25], which block insecure executions, enforce noninterference only for the set of normally terminated traces. These mechanisms may leak information through traces that are blocked, and thus, we characterize them as block-unsafe. By enforcing noninterference for both normally terminated and blocked traces, we show that our mechanism is block-safe, meaning that it does not introduce information leakage through blocking executions.[13]

We prove noninterference with respect to a particular label ℓ ($\ell \neq \bot$) in lattice $\langle \mathcal{L}, \sqsubseteq \rangle$. A value tagged with ℓ may flow only to principals associated with a label at least as restrictive as ℓ. We denote with $\uparrow \ell$ the set of labels that are at least as restrictive as ℓ, and with $\overline{\uparrow \ell}$ the complement of $\uparrow \ell$ with respect to $\mathcal{L}$. Values tagged with labels in $\uparrow \ell$ are considered sensitive with respect to ℓ, and values tagged with labels in $\overline{\uparrow \ell}$ are considered not sensitive with respect to ℓ. Sensitive values with respect to ℓ should not flow to principals associated with labels in $\overline{\uparrow \ell}$.

Noninterference states that changing values of variables tagged with $\uparrow \ell$ should not cause changes of observations made by principals associated with $\overline{\uparrow \ell}$. Equivalently, if two traces Σ_1, Σ_2 of the same command $\mathcal{C}$ start with memories M_1, M_2 (correspondingly) that agree on variables tagged with $\overline{\uparrow \ell}$, without necessarily agreeing on variables tagged with $\uparrow \ell$, then the observations that principals associated with $\overline{\uparrow \ell}$ can make should be the same for these two traces:

$$\Sigma_1|_{\overline{\uparrow \ell}} = \Sigma_2|_{\overline{\uparrow \ell}}.$$

By proving noninterference with respect to observations, which are defined based on fixed and redefined labels on variables, we show that our mechanism

[13] This previous work [9, 4, 12, 13, 28, 25] was proving *termination-insensitive* noninterference (TINI), ignoring all termination channels, both those generated by diverging loops and those introduced by the enforcement mechanism. In this work, we ignore termination channels generated by diverging loops, but we address channels that could have been introduced by our mechanism. It can be argued that we also prone TINI, but for a larger set of traces (a set that contains blocked traces).

does not introduce information leakage through redefining labels for working variables.

The challenging part for formally stating noninterference for our model is to split a memory M into two partitions: partition $M|_{\uparrow\ell}$ contains information regarded sensitive with respect to ℓ, and partition $M|_{\overline{\uparrow\ell}}$ contains information regarded not sensitive with respect to ℓ. Clearly, the former partition contains values that are stored in variables tagged with $\uparrow\ell$, and the latter contains values that are stored in variables tagged with $\overline{\uparrow\ell}$. But a memory, apart from values, it also stores labels, metalabels, *pc*, and *block*.[14] All these components encode information. Thus, we need to decide to which of the two above partitions these components belong; we need to decide their sensitivity with respect to ℓ.

To formally state noninterference, it suffices to define $M|_{\overline{\uparrow\ell}}$, that is, the part of M that must not be influenced by information of level ℓ or above. We explain the construction of $M|_{\overline{\uparrow\ell}}$ by defining the sensitivity of all possible components that may appear in a memory.

Deciding the sensitivity of values and labels is straightforward. The sensitivity of a value in variable x is represented by label $\underline{x}$. So, if $\ell \not\sqsubseteq M(\underline{x})$, then value $M(\underline{x})$ is not sensitive with respect to ℓ, and thus, pair $\langle x,\ M(x)\rangle$ is added in partition $M|_{\overline{\uparrow\ell}}$. The sensitivity of a label $\underline{x}$ is represented by metalabel $\underline{\underline{x}}$. So, if $\ell \not\sqsubseteq M(\underline{\underline{x}})$, then pair $\langle \underline{x},\ M(\underline{x})\rangle$ is added in partition $M|_{\overline{\uparrow\ell}}$.

What is the sensitivity of metalabels? Indeed, the value of a metalabel may depend on sensitive information. Consider the following example:

$$
\begin{array}{l}
\textbf{if } m > 0 \textbf{ then} \\
\quad \textbf{if } h > 0 \textbf{ then } w := 0 \textbf{ else skip end} \\
\textbf{else} \\
\quad w := 1 \\
\textbf{end}
\end{array}
\tag{5}
$$

At the end of the outer conditional command the metalabel of w may be `H` or `M` depending on m. So, the value of $\underline{\underline{w}}$ depends on m, and thus the sensitivity of $\underline{\underline{w}}$ is `M`. In general, we decide whether a pair $\langle \underline{x},\ M(\underline{x})\rangle$ belongs to $M|_{\overline{\uparrow\ell}}$ by just examining the value $M(\underline{\underline{x}})$ of metalabel $\underline{\underline{x}}$. If $\ell \not\sqsubseteq M(\underline{\underline{x}})$, then $M(\underline{x})$ is not influenced by values tagged with $\uparrow\ell$, and thus, $M(\underline{x})$ is regarded not sensitive with respect to ℓ.[15] So, if $\ell \not\sqsubseteq M(\underline{\underline{x}})$, then pair $\langle \underline{\underline{x}},\ M(\underline{\underline{x}})\rangle$ is

[14] The formal definition of a memory M is found in the Appendix.

[15] Assume for contradiction that $M(\underline{x})$ is influenced by a value tagged with $\uparrow\ell$. This

added in $M|_{\overline{\uparrow \ell}}$. Notice that, we do not provide additional machinery to keep track of the sensitivity of metalabels; the sensitivity of a metalabel is given by the metalabel itself.

The *pc* itself can be influenced by sensitive information. Consider the following example:

$$\begin{array}{l} \textbf{if } m > 0 \textbf{ then } w := m \textbf{ else } w := h \textbf{ end}; \\ \textbf{if } w > 0 \textbf{ then } w' := 1 \textbf{ else } w' := 2 \textbf{ end}. \end{array} \tag{6}$$

Here, when entering the second conditional command, *pc* may contain either `M` or `H` depending on the label of w, which depends on m. Again, we use the value of $M(pc).label$ to decide whether *pc* belongs to $M|_{\overline{\uparrow \ell}}$. If $\ell \not\sqsubseteq M(pc).label$, then pair $\langle pc,\ M(pc)\rangle$ is added in $M|_{\overline{\uparrow \ell}}$.

The larger $M|_{\overline{\uparrow \ell}}$ is, the stronger the statement of noninterference becomes. In an effort to construct the largest possible partition $M|_{\overline{\uparrow \ell}}$, even when $\ell \sqsubseteq M(pc).label$, we include in $M|_{\overline{\uparrow \ell}}$ parts of *pc* that do not depend on sensitive information relative to ℓ. Specifically, if the label of a substack of *pc* is not more restrictive than ℓ, then it is not influenced by sensitive information relative to ℓ. We write $pc.pop^n$ to represent a substack of *pc*, created by popping the n top elements of *pc*. Here, n may range from 1 to the size of *pc*. Thus, if $\ell \not\sqsubseteq M(pc).pop^n.label$, then $\langle pc.pop^n,\ M(pc).pop^n\rangle$ is added in $M|_{\overline{\uparrow \ell}}$.

Finally, the value of *block* may depend on sensitive information, too. In the following example:

$$\begin{array}{l} \textbf{if } m > 0 \textbf{ then} \\ \quad \textbf{if } h > 0 \textbf{ then } w := 0 \textbf{ else skip end} \\ \textbf{else} \\ \quad w := 1 \\ \textbf{end} \\ a := w \end{array} \tag{7}$$

the value of *block* that is considered for checking assignment $a := w$ is either `M` or `H`, depending on the metalabel of w, which depends on m. Similarly to the previous cases, if $\ell \not\sqsubseteq M(block)$, then $\langle block,\ M(block)\rangle$ is added in $M|_{\overline{\uparrow \ell}}$.

---

means that $M(\underline{\underline{x}})$ is constructed in a conditional command whose guard is tagged with $\uparrow\ell$. But, due to rule (S-Asgn2), $M(\underline{\underline{x}})$ should be more restrictive than ℓ. Contradiction, because $\ell \not\sqsubseteq M(\underline{\underline{x}})$.

Consequently, $M|_{\overline{\uparrow \ell}}$ is defined as follows:

$$
\begin{aligned}
M|_{\overline{\uparrow \ell}} = & \{\langle x,\ M(x)\rangle \mid \ell \not\sqsubseteq M(\underline{x})\} \cup && (8)\\
& \{\langle \underline{x},\ M(\underline{x})\rangle \mid \ell \not\sqsubseteq M(\underline{x})\} \cup && (9)\\
& \{\langle \underline{\underline{x}},\ M(\underline{\underline{x}})\rangle \mid \ell \not\sqsubseteq M(\underline{\underline{x}})\} \cup && (10)\\
& \{\langle pc,\ M(pc)\rangle \mid \ell \not\sqsubseteq M(pc).label\} \cup && (11)\\
& \{\langle pc.pop^n,\ M(pc).pop^n\rangle && (12)\\
& \qquad \mid \ell \not\sqsubseteq M(pc).pop^n.label\} \cup \\
& \{\langle block,\ M(block)\rangle \mid \ell \not\sqsubseteq M(block)\}. && (13)
\end{aligned}
$$

Notice that labels of anchor variables always belong to $M|_{\overline{\uparrow \ell}}$. This is because metalabels of anchor variables are always $\bot$ (see Section 3), and thus condition $\ell \not\sqsubseteq M(\underline{x})$ in line (9), where $\ell \neq \bot$, always holds for anchor variables.

We prove that programs executed under our operational semantics satisfy noninterference. The Theorem below considers two traces of the same command $\mathcal{C}$, which start from memories M and M' that are *properly initialized*. A memory M is properly initialized if and only if:

- $\forall x.M(\underline{x}) = \bot$,
- $M(pc)$ is empty, and
- $M(block) = \bot$.

Theorem 1. *Consider a label ℓ, a command $\mathcal{C}$, and properly initialized memories M, M'.*
If $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, $\Sigma = \langle \mathcal{C},\ M\rangle \xrightarrow{*} M_t$, and $\Sigma' = \langle \mathcal{C},\ M'\rangle \xrightarrow{*} M'_t$, then $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}}$.

The proof of Theorem 1 is found in the Appendix.[16] This proof involves a lemma (Lemma 4) worth mentioning. This lemma states that, if a command is executed on a properly initialized memory, then, during execution, the label of a variable is always more restrictive than the metalabel of that variable:

$$M(\underline{x}) \sqsubseteq M(\underline{x}). \tag{14}$$

This means that the value of a variable is always more sensitive than the label of that variable. We will see, in Section 5.2, how this lemma is used for deriving a simpler enforcement mechanism for `H` and `L` labels.

[16] Because M and M' are properly initialized, deciding $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$ for the last four subsets of $M|_{\overline{\uparrow \ell}}$ (i.e., lines (10), (11), (12), and (13)) and $M'|_{\overline{\uparrow \ell}}$ is trivial. This triviality vanished once we try to prove the inductive case.

5 Discussion

5.1 Hybrid Enforcement

Some of the run-time overhead introduced by our mechanism can be eliminated, if we statically preprocess programs. Instead of checking dynamically before every assignment to anchor variables if labels are violated, the preprocessing stage would decide which assignments need to be checked dynamically, by inserting the proper check, and which assignments are always secure, in which case there is no need for checking. The more precise information about labels tagging variables this stage collects, the more excessive dynamic checks would be eliminated.

A preprocessing stage would also eliminate the run-time overhead of deciding whether labels, metalabels, and variable *block* need to be silently redefined at the end of conditional commands. Specifically, function *targetWVar* and *anchorVar* would be called during this preprocessing stage. The program code could then be augmented with updates to particular labels and metalabels that should be performed at run-time. This implies that the resulting mechanism would not need to keep track of V and d, simplifying the structure of *pc*. Consequently, the existence of a preprocessing stage would leave the dynamic analysis with only updating labels and checking suspicious assignments to anchor variables.

5.2 Enforcing labels `H` and `L`

We show how to derive a simplified, true flow-sensitive and block-safe enforcement mechanism for two-element lattice $\langle\{\mathtt{H},\mathtt{L}\},\ \sqsubseteq\rangle$, with $\mathtt{L} \sqsubseteq \mathtt{H}$, from our mechanism. Previous work [7] on block-safe mechanism for `H` and `L` labels did not provide true flow-sensitivity.

When our mechanism enforces only labels `H` and `L`, redefined labels of working variables do not depend on sensitive values (i.e., values tagged with `H`). Suppose we want to differentiate the label of a working variable between `H` and `L`, based on a variable tagged with `H`. Consider the following example:

$$\textbf{if } h' > 0 \textbf{ then } w := h \textbf{ else } w := l \textbf{ end}.$$

The goal is to encode sensitive information $h' > 0$ in the label of w. However, using our mechanism, the label of w will be always `H` at the end of the conditional command (if $h' > 0$, then $\underline{w}$ becomes $\underline{h'} \sqcup \underline{h}$, which is `H`, and if $h' \not> 0$, then $\underline{w}$ becomes $\underline{h'} \sqcup \underline{l}$, which is again `H`). This example indicates that under our enforcement mechanism, redefining labels for working variables

$$\text{(S'-Asgn1)}\ \frac{v = M(\mathcal{E}) \qquad M(\underline{\mathcal{E}}) \sqcup M(pc).label \sqcup block' \sqsubseteq M(\underline{a}) \qquad block' = M(block) \sqcup M(pc).label}{\langle a := \mathcal{E},\ M\rangle \rightarrow \langle \textbf{stop},\ M[a \mapsto v, block \mapsto block']\rangle}$$

$$\text{(S'-Asgn2)}\ \frac{v = M(\mathcal{E}) \qquad \ell = M(\underline{\mathcal{E}}) \sqcup M(pc).label}{\langle w := \mathcal{E},\ M\rangle \rightarrow \langle \textbf{stop},\ M[w \mapsto v, \underline{w} \mapsto \ell]\rangle}$$

Table 2: Simplified rules for enforcing only labels `H` and `L`.

does not depend on sensitive information, when only two labels are used. Thus, there is no need to use metalabels to capture the sensitivity of labels.

Here is another reason why metalabels are not needed for enforcing `H` and `L` labels in a true flow-sensitive and block-safe way. Recall that one of the lemmata used to prove Theorem 1 dictates: $M(\underline{x}) \sqsubseteq M(\underline{x})$. If $M(\underline{x})$ is redefined based on sensitive information, then $M(\underline{x})$ ought to be `H`. But if $M(\underline{x})$ is `H`, then from the above restrictiveness relation $M(\underline{x})$ should necessarily be `H`. So, if $M(\underline{x})$ is computed based on sensitive information, then the value of $M(\underline{x})$ is always `H`; $M(\underline{x})$ cannot change when this sensitive information changes. Thus, label $M(\underline{x})$ cannot be used to leak sensitive information. Consequently, there is no need to use metalabels to capture the sensitivity of labels.

A true flow-sensitive and block-safe enforcement mechanism for `H` and `L` is derived from our mechanism by omitting the use of metalabels. Figure 2 contains which rules from Figure 1 should be modified and how. Variable *block* is still needed to track the sensitivity of assignment checks. In this case, the sensitivity of a check is simply the sensitivity of the context in which this check is performed. So, *block* is updated with the value of $M(pc).label$, whenever a check on labels is performed. We prove in the Appendix that if rules (S-Asgn1) and (S-Asgn2) are substituted with (S'-Asgn1) and (S'-Asgn2), then the resulting operational semantics satisfy Theorem 1, meaning that the resulting mechanism is block-safe. Notice that the derived mechanism is no longer block-safe when trying to enforce labels from an arbitrary lattice.

6 Related Work

In this paper, we presented a dynamic mechanism that blocks insecure executions. Our mechanism is block-safe, because it does not leak informa-

FSD mechanisms	Multilevel	True flow-sensitive	Block-safe
[9],[4],[12], [13],[28],[25]	✓	✓	×
[5],[21]	×	✓	×
[7]	×	×	✓
[1]	✓	×	✓
[23],[12]	×	✓	modify, skip
[11]	✓	✓	diverge
[24]	×	✓	diverge
[29]	✓	×	tracking
[19],[20]	×	✓	tracking
[8]	✓	✓	tracking

Table 3: Comparison of related work on FSD mechanisms

tion when blocking execution, true flow-sensitive, and *multilevel*, because it enforces multiple labels (instead of only `H` and `L`). In past work, enforcement mechanisms had a (strict) subset of these three attributes (i.e., block-safe, true flow-sensitive, and multilevel). Table 3 summarizes previous language-based, FSD mechanisms for information flow control in terms of these attributes. In Table 3, we also include hybrid mechanisms, which first statically analyze programs and then monitor their execution. If a mechanism has the attribute of the corresponding column, we mark it with ✓, otherwise we mark it with ×. When attribute "Block-safe" is not applicable for a mechanism (because the mechanism does not block executions), we clarify the action that is taken by that mechanism to address insecure commands (i.e., "modify", "skip", or "diverge"). We write "tracking", if a mechanism does not take any action, and it simply updates variables with proper labels.

Some of the entries of Table 3 (i.e, [9], [4],[13], [28], [5],[21]) that are not block-safe (marked with × at "Block-safe" column) are based on the techniques of *no-sensitive-upgrade* (NSU) and *permissive-upgrade* (PU). Enforcement mechanisms that use these techniques have been examined by Bielova and Rezk [10], who characterized these mechanisms as not *termination aware*.[17] The authors use knowledge-based semantics to show that the

[17]The authors actually use the term *termination aware noninterferent* (TANI) mecha-

attacker's knowledge may increase when programs are executed under these mechanisms (comparing to the attacker's knowledge before running the program). We believe our mechanism is termination aware. The authors of this paper also compare the *transparency* and *precision* of several mechanisms. In future work, we plan to compare the transparency and precision of our mechanism with respect to the mechanisms considered in that paper.

Secure multi-execution (SME) [17] is a flow-sensitive dynamic mechanism that enforces information flow labels by simultaneously executing the same program as many times as the number of labels. An execution that corresponds to a label ℓ sees the actual values, when these values are tagged with labels at most as restrictive as ℓ, and it sees dummy values, otherwise. Using *faceted* values [6], which is a tuple of values, each one corresponding to a different label, one can use one execution to simulate the set of executions generated by SME. These two approaches are precise and do not introduce information leakage. However, the run-time overhead they introduce increases with the number of different labels that may be used.

We are not the first to tag labels with labels. In a purely dynamic context, Buiras et al. [11] use fixed labels on labels to capture the implicit flow of information caused by conditional commands. In our paper, we use a light-weight on-the-fly analysis to capture this implicit flow. Then, we use mutable metalabels to provide block-safety. The mechanism presented by Buiras et al. [11] cause insecure executions to diverge instead of blocking. In this mechanism, sensitive information used to decide whether an execution should diverge may be leaked to the termination channel. In this context, the practicality of diverging, instead of blocking, an execution needs to be carefully examined.

Tagging labels with labels have been also used in flow-insensitive, mostly static mechanisms [31], where labels are treated as first-class values. There, it is the programmer's responsibility to come up with the correct labels and the appropriate checks before commands. On the contrary, our mechanism is dynamic, it automatically redefines labels and metalabels for working variables, and it automatically inserts dynamic checks.

Static language-based approaches have been proposed for controlling information flow, too. Volpano et al. [30] first introduced a static type system for information flow control, based on Denning's lattice model [16, 14, 15]. Sabelfeld and Myers [27] examine certain static language-based approaches (in addition to other approaches). Hunt and Sands [22] introduce flow-sensitivity in static mechanisms. Russo and Sabelfeld [26], then, compared

nism.

flow-sensitive static mechanisms with flow-sensitive dynamic mechanisms.

7 Conclusion

This paper presented a dynamic enforcement mechanism for information flow control that is multilevel, true flow-sensitive, and block-safe. Combining these three attributes was challenging and it was not provided by previous work. Our mechanism offers increased permissiveness, due to true flow-sensitivity, without introducing information leakage when blocking executions, due to block-safety.

For block-safety, we employed metalabels along with a variable *block* to keep track of the sensitivity of information used to decide whether an execution should be blocked. We then allowed principals to make observations only when this sensitivity is not violated. We showed that metalabels themselves cannot be used to leak sensitive information, and thus, there is no need to add meta-metalabels.

We proved that programs executed under our mechanism satisfy noninterference. Our threat model allows principals to make observations during the execution of a program, in which case a mechanism that in not block-safe would leak an arbitrary amount of sensitive information. So, we proved that observations made by unauthorized principals throughout normally terminated or blocked traces are not influenced by initial sensitive values.

To the best of our knowledge there has not been proposed an FSD mechanism that is *termination sensitive*, true flow-sensitive and multilevel. So, as a next step, we are planning to extend our mechanism to enforce termination sensitive noninterference, which is noninterference for all traces of a program (i.e., normally terminated, blocked, and diverging traces). Also, noticing that past work on FSD mechanisms does not formally handle arbitrary reclassifications (i.e., declassifications and classifications), we are planning to use our mechanism to enforce richer information flow labels that can specify arbitrary reclassifications. Finally, we would like to examine whether actions "skip" and "modify", which are used by some FSD mechanisms to skip or modify insecure commands, introduce new covert channels. We are inspiring of a theorem stating that "skip" or "modify" actions do not leak sensitive information encoded in labels of working variables, and thus metalabels are not needed to provide noninterference for finite traces.

Acknowledgments

We are deeply thankful to Fred B. Schneider for encouraging us to use the idea of metalabels and for giving us valuable comments on earlier versions of this work.

References

[1] A. Askarov, S. Chong, and H. Mantel. Hybrid monitors for concurrent noninterference. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 137–151, July 2015.

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

[3] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 43–59, July 2009.

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

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[7] A. Bedford, S. Chong, J. Desharnais, and N. Tawbi. A progress-sensitive flow-sensitive inlined information-flow control monitor. In *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings*, pages 352–366, Cham, 2016. Springer International Publishing.

[8] L. Beringer. End-to-end multilevel hybrid information flow control. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pages 50–65, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 15:15–15:24, New York, NY, USA, 2014. ACM.

[10] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 46–67, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[11] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.

[12] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 200–214, July 2010.

[13] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 629–643, New York, NY, USA, 2015. ACM.

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

[15] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[16] D. E. R. Denning. *Secure information flow in computer systems.* PhD thesis, Purdue University, West Lafayette, IN, USA, 1975.

[17] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security*

and Privacy, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[19] G. L. Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop*, 2008.

[20] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 351–365, July 2015.

[21] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 3–18, Washington, DC, USA, 2012. IEEE Computer Society.

[22] S. Hunt and D. Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[23] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06, pages 75–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[24] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Comput. Secur.*, 31(7):827–843, Oct. 2012.

[25] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 146–160, 2011.

[26] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[28] J. F. Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In *ICT Systems Security and Privacy Protection: 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings*, pages 278–292, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[29] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[30] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[31] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Secur.*, 6(2):67–84, Mar. 2007.

A Appendix

Formally, memory M consists of the following mappings:

$$\begin{aligned}
&M_x : \mathcal{V} \to \mathit{int},\\
&M_{\underline{x}} : \{\underline{x} | x \in \mathcal{V}\} \to \mathcal{L},\\
&M_{\underline{\underline{x}}} : \{\underline{\underline{x}} | x \in \mathcal{V}\} \to \mathcal{L},\\
&M_{pc} : \{pc\} \to \mathit{Stacks},\\
&M_{block} : \{block\} \to \mathcal{L}.
\end{aligned}$$

The statement of Proposition 3 below considers two initial states whose memories M, M' satisfy the following predicates:

- $\mathit{healthy}(M) \triangleq \forall x. M(\underline{x}) \sqsubseteq M(\underline{x})$
- $\mathit{consistent}(M, M', \ell) \triangleq$ for all $n \geq 0$,
  - $M(pc).pop^n.top.d \wedge \ell \sqsubseteq M(pc).pop^n.label \Rightarrow$
    $\ell \sqsubseteq M'(block)$ or
    $M'(pc).pop^n.top.d \wedge \ell \sqsubseteq M'(pc).pop^n.label$,
  - $w \in M(pc).pop^n.top.V \wedge \ell \sqsubseteq M(pc).pop^n.label \Rightarrow$
    $\ell \sqsubseteq M'(\underline{w}) \wedge \ell \sqsubseteq M'(\underline{\underline{w}})$ or
    $w \in M'(pc).pop^n.top.\overline{V} \wedge \ell \sqsubseteq M'(pc).pop^n.label$.

The same for M' and M.

The last predicate formalizes the role that d and V play at each element of the *pc* stack. In particular, if d is true in an element of *pc* at one execution, and if the label of *pc* up to that element is more restrictive than ℓ, then it means that the other execution involves dynamic checks in context more restrictive than ℓ, which raises *block* of that second execution to be at least as restrictive as ℓ. In a similar fashion, if w belongs to V in an element of *pc* at one execution, and if the label of *pc* up to that element is more restrictive than ℓ, then it means that the other execution involves assignment to w in context more restrictive than ℓ, which raises $\underline{w}$ and $\underline{\underline{w}}$ of that second execution to be at least as restrictive as ℓ. Notice that if M and M' are properly initialized, then $\mathit{healthy}(M)$, $\mathit{healthy}(M')$, and $\mathit{consistent}(M, M', \ell)$ hold.

Theorem 1. *Consider a label ℓ, a command $\mathcal{C}$, and properly initialized memories M, M'.*
If $M|_{\overline{\uparrow\ell}} = M'|_{\overline{\uparrow\ell}}$, $\Sigma = \langle \mathcal{C},\ M\rangle \xrightarrow{*} M_t$, and $\Sigma' = \langle \mathcal{C},\ M'\rangle \xrightarrow{*} M'_t$, then $\Sigma|_{\overline{\uparrow\ell}} = \Sigma'|_{\overline{\uparrow\ell}}$.

Proof. Memories M and M' are properly initialized, and thus, $\mathit{healthy}(M)$, $\mathit{healthy}(M')$, and $\mathit{consistent}(M, M', \ell)$ hold. So, we can instantiate Proposition 3, with label ℓ, command $\mathcal{C}$, and memories M, M'.
If $M|_{\overline{\uparrow\ell}} = M'|_{\overline{\uparrow\ell}}$, $\Sigma = \langle \mathcal{C},\ M\rangle \xrightarrow{*} M_t$, and $\Sigma' = \langle \mathcal{C},\ M'\rangle \xrightarrow{*} M'_t$, then conclusion [$c$0] of Proposition 3 gives us $\Sigma|_{\overline{\uparrow\ell}} = \Sigma'|_{\overline{\uparrow\ell}}$. □

Lemma 2. *Each observation $\langle x,\ M(x)\rangle$ in $\Sigma|_{\overline{\uparrow\ell}}$ is produced when formula $\exists \ell' \in \overline{\uparrow\ell}.M(\underline{x}) \sqcup M(\mathit{block}) \sqsubseteq \ell'$ holds. An equivalent, and simpler formula (which is used for the proofs) is: $\ell \not\sqsubseteq M(\underline{x}) \sqcup M(\mathit{block})$.*

Proof. $\Rightarrow$: if $\ell \sqsubseteq M(\underline{x}) \sqcup M(\mathit{block})$ then $M(\underline{x}) \sqcup M(\mathit{block}) \in \uparrow\ell$, thus any ℓ' such that $M(\underline{x}) \sqcup M(\mathit{block}) \sqsubseteq \ell'$ belongs to $\uparrow\ell$, and hence is not in $\overline{\uparrow\ell}$.
$\Leftarrow$: if $\ell \not\sqsubseteq M(\underline{x}) \sqcup M(\mathit{block})$ then $\ell' = M(\underline{x}) \sqcup M(\mathit{block}) \in \overline{\uparrow\ell}$ satisfies the left-hand side. □

Proposition 3. *Consider a label ℓ, a residual command $\mathcal{C}^-$, and memories M, M', such that $\mathit{healthy}(M)$, $\mathit{healthy}(M')$, $\mathit{consistent}(M, M', \ell)$.*
Assume $M|_{\overline{\uparrow\ell}} = M'|_{\overline{\uparrow\ell}}$, and $\Sigma = \langle \mathcal{C}^-,\ M\rangle \xrightarrow{*} M_t$ and $\Sigma' = \langle \mathcal{C}^-,\ M'\rangle \xrightarrow{*} M'_t$, where $\langle \mathcal{C}^-_t,\ M_t\rangle$ is the last state of Σ and $\langle \mathcal{C}^-_{t'},\ M'_t\rangle$ is the last state of Σ'. Then:

c0 $\Sigma|_{\overline{\uparrow\ell}} = \Sigma'|_{\overline{\uparrow\ell}}$.

c1 *If* Σ *and* Σ' *terminate normally, then* $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$ *and* $\mathit{consistent}(M_t, M'_t, \ell)$.

c2 *If* Σ' *terminates normally, and* Σ *is blocked due to rule* (S-Asgn1), *and* $\ell \sqsubseteq M_t(\mathit{block}')$, *then* $\ell \sqsubseteq M'_t(\mathit{block})$. *We write* $M_t(\mathit{block}')$ *for* $M_t(\mathit{block}) \sqcup M_t(\underline{\underline{\mathcal{E}}}) \sqcup M_t(\mathit{pc}).\mathit{label}$.

c3 *If* Σ' *terminates normally and* $\ell \not\sqsubseteq M_t(\mathit{pc}).\mathit{label}$, *then there exists* M''_t *such that* $\langle \mathcal{C}^-,\ M' \rangle \xrightarrow{*} \langle \mathcal{C}^-_t,\ M''_t \rangle$ *is a prefix of* Σ' *and* $M_t|_{\overline{\uparrow \ell}} = M''_t|_{\overline{\uparrow \ell}}$.

Proof. Structural induction on $\mathcal{C}^-$.

Case 1. $a := \mathcal{E}$
We first prove the second part of c1; afterwards, we prove c0-c3 by cases.
If both Σ *and* Σ' *terminate,* $M_t(\mathit{pc}) = M(\mathit{pc})$, $M'_t(\mathit{pc}) = M'(\mathit{pc})$, $M_t(\mathit{block}) = M(\mathit{block})$, $M'_t(\mathit{block}) = M'(\mathit{block})$, $\forall w.M_t(\underline{w}) = M(\underline{w})$, $\forall w.M'_t(\underline{w}) = M'(\underline{w})$, $\forall w.M_t(\underline{\underline{w}}) = M(\underline{\underline{w}})$, *and* $\forall w.M'_t(\underline{\underline{w}}) = M'(\underline{\underline{w}})$.
Because $\mathit{consistent}(M, M', \ell)$, *from the above we also get* $\mathit{consistent}(M_t, M'_t, \ell)$.
We also have $M(\underline{a}) = M'(\underline{a})$*: by definition,* $\ell \not\sqsubseteq M(\underline{\underline{a}}) = M'(\underline{\underline{a}}) = \bot$, *which implies* $\underline{a} \in \mathrm{dom}(M|_{\overline{\uparrow \ell}})$, *and because* $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, *we have* $M(\underline{a}) = M'(\underline{a})$.

Case 1.1. *Let* $\ell \not\sqsubseteq M(\underline{a}) = M'(\underline{a})$. *We first prove that the command is executed normally in both memories or blocked in both memories. W.l.o.g., assume that the command is executed normally in* M. *That is,* $M(\underline{\underline{\mathcal{E}}}) \sqcup M(\mathit{pc}).\mathit{label} \sqcup M(\mathit{block}') \sqsubseteq M(\underline{a})$, *where* $M(\mathit{block}') = M(\mathit{block}) \sqcup M(\underline{\underline{\mathcal{E}}}) \sqcup M(\mathit{pc}).\mathit{label}$.
This implies that ℓ *is not less restrictive than any members of the supremum, that is,* $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$, $\ell \not\sqsubseteq M(\mathit{pc}).\mathit{label}$, *and* $\ell \not\sqsubseteq M(\mathit{block}')$*; thus,* $\ell \not\sqsubseteq M(\mathit{block})$ *and* $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$. *Hence the labels involved in these expressions are in* $M|_{\overline{\uparrow \ell}}$.
Then, because $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, *they are also in* $M'|_{\overline{\uparrow \ell}}$ *and the two memories agree on them, that is,* $M(\mathcal{E}) = M'(\mathcal{E})$, $M(\mathit{pc}) = M'(\mathit{pc})$, $M(\mathit{block}) = M'(\mathit{block})$, $M(\underline{\underline{\mathcal{E}}}) = M'(\underline{\underline{\mathcal{E}}})$, *and* $M(\underline{\mathcal{E}}) = M'(\underline{\mathcal{E}})$.
So, $M(\mathit{block}') = M'(\mathit{block}')$, *and* $M(\underline{a}) = M'(\underline{a})$ *by the argument above. Thus,* $M'(\underline{\mathcal{E}}) \sqcup M'(\mathit{pc}).\mathit{label} \sqcup M'(\mathit{block}') \sqsubseteq M'(\underline{a})$, *as wanted.*

Case 1.1.1. *The command is executed normally in both memories. Then* $\Sigma = \langle a := \mathcal{E},\ M \rangle \to \langle \mathbf{stop},\ M[a \mapsto M(\mathcal{E}), \mathit{block} \mapsto M(\mathit{block}')] \rangle$, *and* $\Sigma' = \langle a := \mathcal{E},\ M' \rangle \to \langle \mathbf{stop},\ M'[a \mapsto M'(\mathcal{E}), \mathit{block} \mapsto M'(\mathit{block}')] \rangle$.
We have $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \langle a,\ M(\mathcal{E}) \rangle$ *because* $M(\mathcal{E}) = M'(\mathcal{E})$. *Thus,* [c0] *holds.*
Also, we have $M_t(a) = M'_t(a)$ *and* $M_t(\mathit{block}) = M'_t(\mathit{block})$. *Hence* [c1], [c3].
Finally, [c2] *is trivially true.*

Case 1.1.2. *Both traces are blocked. Then* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$, $[c0]$. *Finally,* $[c1], [c2], [c3]$ *are trivially true.*

Case 1.2. *Let* $\ell \sqsubseteq M(\underline{a}) = M'(\underline{a})$.
Then $\ell \sqsubseteq M_t(\underline{a}) \sqcup M_t(block)$, *and hence, by Lemma 2, we have* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. $[c0]$
Because $\ell \sqsubseteq M(\underline{a}) = M_t(\underline{a})$, *we have* $a \notin M_t|_{\overline{\uparrow \ell}}$, *and hence, we only have to check variable block to prove* $[c1]$.

Case 1.2.1. *Let* $\ell \not\sqsubseteq M_t(block) = M(block) \sqcup M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label$ *and both* Σ, Σ' *terminate.*
Then, $\ell \not\sqsubseteq M(block)$, $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$, $\ell \not\sqsubseteq M(pc).label$.
Because $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, $\ell \not\sqsubseteq M'(block)$, $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$, *and* $\ell \not\sqsubseteq M'(pc).label$.
Also, $M(block) = M'(block)$, $M(\underline{\underline{\mathcal{E}}}) = M'(\underline{\underline{\mathcal{E}}})$, *and* $M(pc).label = M'(pc).label$.
So, $M'_t(block) = M_t(block)$ *and* $\ell \not\sqsubseteq M'_t(block)$.
Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$
$[c2]$ *is trivially true.*

Case 1.2.2. *Let* $\ell \sqsubseteq M_t(block)$ *and both* Σ, Σ' *terminate.*
By symmetry of the preceding case, $\ell \sqsubseteq M'_t(block)$.
So, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$, *since block is not in their domain.* $[c1], [c3]$
$[c2]$ *is trivially true.*

Case 1.2.3. *Let* Σ' *terminate and* Σ *does not.*
$[c1]$ *is trivially true. We have:*
$M'_t(block) = M'(block) \sqcup M'(\underline{\underline{\mathcal{E}}}) \sqcup M'(pc).label$ *and*
$M_t(block') = M(block) \sqcup M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label$.
$\ell \not\sqsubseteq M'_t(block) \Rightarrow$
$\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$, $\ell \not\sqsubseteq M'(pc).label$, *and* $\ell \not\sqsubseteq M'(block) \Rightarrow$ *(because* $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$*)*
$\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$, $\ell \not\sqsubseteq M(pc).label$, *and* $\ell \not\sqsubseteq M(block)$,
$M(\underline{\underline{\mathcal{E}}}) = M'(\underline{\underline{\mathcal{E}}})$, $M(pc) = M'(pc)$, *and* $M(block) = M'(block)$.
So $\ell \not\sqsubseteq M_t(block)$, *as wanted. So,* $M_t(block') = M_t(block)$, *and thus,* $\ell \not\sqsubseteq M_t(block')$.
Thus, $\ell \sqsubseteq M'_t(block) \Rightarrow \ell \sqsubseteq M_t(block')$. $[c2]$
We have $\langle a := \mathcal{E},\ M' \rangle \leq \Sigma'$, *so* $M''_t = M'$, *and* $M''_t|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}} = M|_{\overline{\uparrow \ell}} = M_t|_{\overline{\uparrow \ell}}$. $[c3]$

Case 2. $w := \mathcal{E}$
$\Sigma = \langle w := \mathcal{E},\ M \rangle \rightarrow \langle \mathbf{stop},\ M_t \rangle$
$\Sigma' = \langle w := \mathcal{E},\ M' \rangle \rightarrow \langle \mathbf{stop},\ M'_t \rangle$
$[c2]$ *is trivially true.*
We prove $consistent(M_t, M'_t, \ell)$.

We have $M_t(pc) = M(pc)$, $M'_t(pc) = M'(pc)$, $M_t(block) = M(block)$, $M'_t(block) = M'(block)$. *Only working variable* w *changes.*
Let $w \in M_t(pc).pop^n.top.V$ *and* $\ell \sqsubseteq M_t(pc).pop^n.label$.
Then $\ell \sqsubseteq M_t(pc).label$. *So,* $\ell \sqsubseteq M(pc).label$. *So,* $\ell \sqsubseteq M'(pc).label$.
Thus, $\ell \sqsubseteq M'(pc).label \sqsubseteq M'_t(\underline{w}), M'_t(\underline{\underline{w}})$.
So, consistent(M_t, M'_t, ℓ).

Case 2.1. *Let* $\ell \not\sqsubseteq M_t(\underline{w}) \sqcup M_t(block)$.
Then, $\ell \not\sqsubseteq M(\underline{\mathcal{E}})$, $\ell \not\sqsubseteq M(pc).label$, *and* $\ell \not\sqsubseteq M(block)$.
So, $\ell \not\sqsubseteq M'(\underline{\mathcal{E}})$, $\ell \not\sqsubseteq M'(pc).label$, *and* $\ell \not\sqsubseteq M'(block)$.
So, due to healthy(M) *and healthy*(M'), *we have* $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$.
Thus M *and* M' *agree on* $block, \mathcal{E}, \underline{\mathcal{E}}, \underline{\underline{\mathcal{E}}}$ *and* pc.
Thus M_t *and* M'_t *agree on* $\underline{w}$ *and* $\underline{\underline{w}}$, *and* $M_t(w) = M(\mathcal{E}) = M'(\mathcal{E}) = M'_t(w)$.
So, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$
Also, $\ell \not\sqsubseteq M_t(\underline{x}) \sqcup M_t(block)$. *So,* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \langle w,\ M(\mathcal{E})\rangle$. $[c0]$

Case 2.2. *Let* $\ell \sqsubseteq M_t(\underline{w}) \sqcup M_t(block)$.
By symmetry of the preceding case, $\ell \sqsubseteq M'_t(\underline{w}) \sqcup M'_t(block)$.
So, $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. $[c0]$

Case 2.2.1. *Let* $\ell \not\sqsubseteq M_t(\underline{\underline{w}}) = M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label$.
Then, $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M(pc).label$.
So, $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M'(pc).label$.
Also, $M(\underline{\mathcal{E}}) = M'(\underline{\mathcal{E}})$, $M(\underline{\underline{\mathcal{E}}}) = M'(\underline{\underline{\mathcal{E}}})$, *and* $M(pc) = M'(pc)$.
So, $\ell \not\sqsubseteq M'_t(\underline{\underline{w}}) = M_t(\underline{\underline{w}})$ *and* $M'_t(\underline{\underline{w}}) = M_t(\underline{\underline{w}})$.
If $\ell \sqsubseteq M_t(\underline{\underline{w}})$, *then* $\ell \sqsubseteq M'_t(\underline{w})$, *and thus* $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$
If $\ell \not\sqsubseteq M_t(\underline{\underline{w}}) = M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label$, *then* $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M(pc).label$.
So, $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$, *and thus,* $M(\mathcal{E}) = M'(\mathcal{E})$, *and thus* $M_t(w) = M'_t(w)$.
Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$

Case 2.2.2. *Let* $\ell \sqsubseteq M_t(\underline{\underline{w}})$.
Then $\ell \sqsubseteq M'_t(\underline{\underline{w}})$, *due to* $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, *and thus* $\ell \sqsubseteq M_t(\underline{w}), M'_t(\underline{w})$, *due to healthy*$(M)$ *and healthy*(M'). *Thus,* $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$

Case 3. $\mathcal{C}_1^-; \mathcal{C}_2^-$
We first prove the second part of c1; afterwards, we prove c0-c3 by cases.
Assume both Σ *and* Σ' *terminate.*
Consider $\Sigma = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M_1\rangle \xrightarrow{*} \langle \mathbf{stop},\ M_t\rangle$ *and*
$\Sigma' = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M'_1\rangle \xrightarrow{*} \langle \mathbf{stop},\ M'_t\rangle$.
From hypothesis, consistent(M, M', ℓ).
From IH on $\mathcal{C}_1^-$, *consistent*(M_1, M'_1, ℓ). *From Lemma 4 and IH on* $\mathcal{C}_2^-$, *consistent*(M_t, M'_t, ℓ).

Case 3.1. *$\mathcal{C}_1^-$ terminates normally in Σ and Σ'*
Let $\Sigma = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M_2\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t\rangle$ *and*
$\Sigma' = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t'}^-,\ M_t'\rangle$.
Consider
$\Sigma_1 = \langle \mathcal{C}_1^-,\ M\rangle \xrightarrow{*} \langle \mathbf{stop},\ M_2\rangle$, $\Sigma_2 = \langle \mathcal{C}_2^-,\ M_2\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t\rangle$,
$\Sigma_1' = \langle \mathcal{C}_1^-,\ M'\rangle \xrightarrow{*} \langle \mathbf{stop},\ M_2'\rangle$, $\Sigma_2' = \langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t'}^-,\ M_t'\rangle$.
From IH on $\mathcal{C}_1^-$, *we get* $\Sigma_1|_{\overline{\uparrow \ell}} = \Sigma_1'|_{\overline{\uparrow \ell}}$, $M_2|_{\overline{\uparrow \ell}} = M_2'|_{\overline{\uparrow \ell}}$, *and consistent*$(M_2, M_2', \ell)$.
From IH on $\mathcal{C}_2^-$ *and Lemma 4, we get* $\Sigma_2|_{\overline{\uparrow \ell}} = \Sigma_2'|_{\overline{\uparrow \ell}}$.
So, $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}}$. [c0]
If Σ *and* Σ' *terminate normally, then* Σ_2 *and* Σ_2' *terminate normally, and thus* $M_t|_{\overline{\uparrow \ell}} = M_t'|_{\overline{\uparrow \ell}}$. [c1]
If Σ' *terminates normally and* $\ell \sqsubseteq M_t(block')$, *then* Σ_2' *terminates normally and* $\ell \sqsubseteq M_t(block')$. *So,* $\ell \sqsubseteq M_t'(block)$. [c2]
If Σ' *terminates normally and* $\ell \not\sqsubseteq M_t(pc).label$, *then* Σ_2' *terminates normally and* $\ell \not\sqsubseteq M_t(pc).label$.
So, by IH, there is some $\langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t''\rangle \leq \Sigma_2'$ *and* $M_t|_{\overline{\uparrow \ell}} = M_t''|_{\overline{\uparrow \ell}}$.
Thus $\langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t''\rangle \leq \Sigma'$ *and* $M_t|_{\overline{\uparrow \ell}} = M_t''|_{\overline{\uparrow \ell}}$. [c3]

Case 3.2. *$\mathcal{C}_1^-$ blocked in both Σ and Σ'*
Let $\Sigma = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-; \mathcal{C}_2^-,\ M_t\rangle$ *and*
$\Sigma' = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_{1t'}^-; \mathcal{C}_2^-,\ M_t'\rangle$.
We consider
$\Sigma_1 = \langle \mathcal{C}_1^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-,\ M_t\rangle$, $\Sigma_1' = \langle \mathcal{C}_1^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_{1t'}^-,\ M_t'\rangle$.
From IH on $\mathcal{C}_1^-$, Σ_1, Σ_1', *we get* $[c0], [c1], [c2], [c3]$.

Case 3.3. *$\mathcal{C}_1^-$ is blocked in Σ, terminates normally in Σ'*
Let $\Sigma = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-; \mathcal{C}_2^-,\ M_t\rangle$ *and*
$\Sigma' = \langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M'\rangle \xrightarrow{*} \langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t'\rangle$.
$[c1]$ *is trivially true.*
Consider
$\Sigma_1 = \langle \mathcal{C}_1^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-,\ M_t\rangle$,
$\Sigma_1' = \langle \mathcal{C}_1^-,\ M'\rangle \xrightarrow{*} \langle \mathbf{stop},\ M_2'\rangle$, $\Sigma_2' = \langle \mathcal{C}_2^-,\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_{2t}^-,\ M_t'\rangle$.
Also:
Σ' *terminates normally and* $\ell \sqsubseteq M_t(block') \Rightarrow$
Σ_1' *terminates normally and* $\ell \sqsubseteq M_t(block') \Rightarrow \ell \sqsubseteq M_2'(block)$
$\Rightarrow \ell \sqsubseteq M_t'(block)$, *from Lemma 7.* [c2]
Also:
Σ' *terminates normally and* $\ell \not\sqsubseteq M_t(pc).label \Rightarrow$
Σ_1' *terminates normally and* $\ell \not\sqsubseteq M_t(pc).label \Rightarrow$

there exists $\langle \mathcal{C}_1^-,\ M' \rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-,\ M_t'' \rangle \leq \Sigma_1'$ *and* $M_t|_{\overline{\uparrow \ell}} = M_t''|_{\overline{\uparrow \ell}}$.
So, there exists $\langle \mathcal{C}_1^-; \mathcal{C}_2^-,\ M' \rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-; \mathcal{C}_2^-,\ M_t'' \rangle \leq \Sigma'$ *and* $M_t|_{\overline{\uparrow \ell}} = M_t''|_{\overline{\uparrow \ell}}$. [$c3$]
From IH on $\mathcal{C}_1^-$, *we get* $\Sigma_1|_{\overline{\uparrow \ell}} = \Sigma_1'|_{\overline{\uparrow \ell}}$.
To prove $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}}$, *it suffices that* $\Sigma_2'|_{\overline{\uparrow \ell}} = \epsilon$.
So we prove $\Sigma_2'|_{\overline{\uparrow \ell}} = \epsilon$.
Trace Σ_1 *is blocked due to* (S-Asgn1), *so* $\mathcal{C}_{1t}^- = a := \mathcal{E}; \ldots$

Case 3.3.1. *Let* $\ell \sqsubseteq M_t(pc).label$.
Then $\ell \sqsubseteq M_t(block')$. *From IH*[$c2$] *on* $\mathcal{C}_1^-$, Σ_1, Σ_1', $\ell \sqsubseteq M_2'(block)$.
So, using Lemma 7, every $a := \mathcal{E}$ *in* $\mathcal{C}_2^-$ *is executed in a memory* M'' *only if* $\ell \sqsubseteq M_2'(block) \sqsubseteq M''(\underline{a})$, *and for every* $w := \mathcal{E}$ *in* $\mathcal{C}_2^-$, $\ell \sqsubseteq M_2'(block) \sqsubseteq M''(\underline{w}) \sqcup M''(block)$.
So, $\Sigma_2|_{\overline{\uparrow \ell}} = \epsilon$. [$c0$]

Case 3.3.2. *Let* $\ell \not\sqsubseteq M_t(pc).label$.
From IH[$c3$] *on* $\mathcal{C}_1^-$, Σ_1, Σ_1', *there exists* $\langle \mathcal{C}_1^-,\ M' \rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-,\ M_1' \rangle \leq \Sigma_1'$ *and* $M_t|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$.
So, $\ell \not\sqsubseteq M_1'(pc).label$ *and* $M_t(pc) = M_1'(pc)$.
Because Σ_1 *was blocked, we have* $M_t(\underline{\mathcal{E}}) \sqcup M_t(pc).label \sqcup M_t(block) \sqcup M_t(\underline{\underline{\mathcal{E}}}) \not\sqsubseteq M_t(\underline{a})$. *Since the equality is satisfied in* Σ_1', *it means that some of the values of* $\underline{\mathcal{E}}$, *block,* $\underline{\underline{\mathcal{E}}}$ *are different in* M_t *and* M_1'.
If $M_t(\underline{\mathcal{E}}) \neq M_1'(\underline{\mathcal{E}})$, *then* $\ell \sqsubseteq M_1'(\underline{\mathcal{E}})$. *So,* $\ell \sqsubseteq M_1'(block)$, *thus from Lemma 7,* $\ell \sqsubseteq M_2'(block)$.
So, following the arguments of case 3.3.1, $\Sigma_2|_{\overline{\uparrow \ell}} = \epsilon$. [$c0$]
The same argument applies if $M_t(\underline{\underline{\mathcal{E}}}) \neq M_1'(\underline{\underline{\mathcal{E}}})$ *or* $M_t(block) \neq M_1'(block)$.

Case 4. if $\mathcal{E}$ then $\mathcal{C}_1$ else $\mathcal{C}_2$ end

Case 4.1. *Let* $\ell \not\sqsubseteq M(\underline{\mathcal{E}})$.
Then $\ell \not\sqsubseteq M'(\underline{\mathcal{E}})$ *and* $M(\mathcal{E}) = M'(\mathcal{E})$.
So, Σ *and* Σ' *get the same branch, say* $\mathcal{C}_1$.
$\Sigma = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M \rangle \rightarrow \langle \mathcal{C}_1; \textbf{end},\ M_1 \rangle \xrightarrow{*} \langle \mathcal{C}_t,\ M_t \rangle$,
$\Sigma' = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M' \rangle \rightarrow \langle \mathcal{C}_1; \textbf{end},\ M_1' \rangle \xrightarrow{*} \langle \mathcal{C}_t',\ M_t' \rangle$.

Case 4.1.1. *Let* $\ell \not\sqsubseteq M_1(pc).label$.
Then $\ell \not\sqsubseteq M(pc).label$, *because* $M(pc).label \sqsubseteq M_1(pc).label$.
So, $\ell \not\sqsubseteq M'(pc).label$ *and* $M(pc) = M'(pc)$.
We have $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$, *because* $M(\underline{\underline{\mathcal{E}}}) \sqsubseteq M(\underline{\mathcal{E}})$ *and* $M'(\underline{\underline{\mathcal{E}}}) \sqsubseteq M'(\underline{\mathcal{E}})$.
So, $M(\underline{\mathcal{E}}) = M'(\underline{\mathcal{E}})$.
Thus, $M_1(pc) = M_1'(pc)$, *because also* d *and* V *are the same for* Σ *and* Σ'.

Thus, $M_1|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$ *and* $\ell \not\sqsubseteq M_1'(pc).label$*. Because* $\ell \not\sqsubseteq M_1(pc).label$ *and* $\ell \not\sqsubseteq M_1'(pc).label$*,*
we get $consistent(M_1, M_1', \ell)$*.*

Case 4.1.2. *Let* $\ell \sqsubseteq M_1(pc).label$*.*
Then $\ell \sqsubseteq M_1'(pc).label$*, by the symmetry of the above arguments.*
Let $\ell \not\sqsubseteq M_1(pc).pop.label$*.*
So, $\ell \not\sqsubseteq M(pc).label$*. Thus,* $\ell \not\sqsubseteq M'(pc).label$ *and* $M(pc) = M'(pc)$*.*
So, $\ell \not\sqsubseteq M_1'(pc).pop.label$ *and* $M_1(pc).pop = M_1'(pc).pop$*.*
Let also $\ell \not\sqsubseteq M_1(pc).pop^n.label$ *with* $n > 1$*.*
Then $\ell \not\sqsubseteq M(pc).pop^{n-1}.label = M_1(pc).pop^n.label$*.*
So, $M'(pc).pop^{n-1} = M(pc).pop^{n-1}$*.*
Thus, $M_1'(pc).pop^n = M_1(pc).pop^n$*.*
Thus, $M_1|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$*.*
Now, we prove that $consistent(M_1, M_1', \ell)$ *holds.*
In the definition of $consistent(M_1, M_1', \ell)$*, first consider* $n = 0$*.*
Assume $M_1(pc).top.d$ *holds.*
Because both traces take branch $\mathcal{C}_1$*, we have:*
$M_1(pc).top.d = M_1'(pc).top.d = anchorVar(\mathcal{C}_2)$*.*
So, $M_1'(pc).top.d$ *holds, too.*
Also, from the hypothesis of this subcase we have $\ell \sqsubseteq M_1(pc).label$ *and* $\ell \sqsubseteq M_1'(pc).label$*.*
For $n > 0$*, we have:*
$M_1(pc).pop^n.top.d \wedge \ell \sqsubseteq M_1(pc).pop^n.label$
$\Rightarrow M(pc).pop^{n-1}.top.d \wedge \ell \sqsubseteq M(pc).pop^{n-1}.label$
$\Rightarrow \ell \sqsubseteq M'(block) \vee$
$M'(pc).pop^{n-1}.top.d \wedge \ell \sqsubseteq M'(pc).pop^{n-1}.label$
$\Rightarrow \ell \sqsubseteq M_t'(block) \vee$
$M_t'(pc).pop^n.top.d \wedge \ell \sqsubseteq M_t'(pc).pop^n.label$*.*
We work similarly for the second component (working variables) of the definition of consistent.
So, $consistent(M_1, M_1', \ell)$*.*

In both cases, $healthy(M_1)$ *and* $healthy(M_1')$ *hold, because* $healthy(M)$*,* $healthy(M')$ *and no label or metalabel changed along the transition to* $\mathcal{C}_1; \mathbf{end}$*. Consequently, in both cases we can use Lemma 4, apply the IH on* $\mathcal{C}_1; \mathbf{end}$*, and get* $[c0], [c1], [c2], [c3]$*.*

Case 4.2. *Let* $\ell \sqsubseteq M(\underline{\mathcal{E}})$*.*
Then $\ell \sqsubseteq M'(\underline{\mathcal{E}})$*.*
So, Σ *and* Σ' *may get different branches:*

$\Sigma = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M\rangle \rightarrow \langle \mathcal{C}_1; \textbf{end},\ M_1\rangle \xrightarrow{*} \langle \mathcal{C}_t,\ M_t\rangle$,
$\Sigma' = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M'\rangle \rightarrow \langle \mathcal{C}_2; \textbf{end},\ M'_2\rangle \xrightarrow{*} \langle \mathcal{C}'_t,\ M'_t\rangle$.
In both traces, $\ell \sqsubseteq M_1(pc).label, M'_2(pc).label$.
From Lemma 9 we get $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. [c0]
Assume that both traces terminate:
$\Sigma = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M\rangle \rightarrow \langle \mathcal{C}_1; \textbf{end},\ M_1\rangle \xrightarrow{*} \langle \textbf{end},\ M_{1t}\rangle \rightarrow \langle \textbf{stop},\ M_t\rangle$,
$\Sigma' = \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M'\rangle \rightarrow \langle \mathcal{C}_2; \textbf{end},\ M'_2\rangle \xrightarrow{*} \langle \textbf{end},\ M'_{2t}\rangle \rightarrow \langle \textbf{stop},\ M'_t\rangle$.
We want to show that $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$.
Case *pc:* *From Lemma 6, the value of pc is restored at the exit of the conditional, that is,* $M_t(pc) = M(pc)$ *and* $M'_t(pc) = M'(pc)$.
If $\ell \not\sqsubseteq M_t(pc).label = M(pc).label$, *then* $\ell \not\sqsubseteq M'(pc).label$, *and* $M(pc) = M'(pc)$, *and thus,* $M_t(pc) = M'_t(pc)$.
If $\ell \not\sqsubseteq M_t(pc).pop^n().label$, *for* $n \geq 1$,
then $\ell \not\sqsubseteq M(pc).pop^n().label$, *then* $\ell \not\sqsubseteq M'(pc).pop^n().label$, *and* $M(pc).pop^n() = M'(pc).pop^n()$,
and then $M_t(pc).pop^n() = M'_t(pc).pop^n()$.
Case *block:*

- *If* $anchorVar(\mathcal{C}_1)$ *and* $anchorVar(\mathcal{C}_2)$, *then from Lemma 8,* $\ell \sqsubseteq M_{1t}(block)$ *and* $\ell \sqsubseteq M'_{2t}(block)$.
  From Lemma 7, $\ell \sqsubseteq M_t(block), M'_t(block)$.

- *If* $\neg anchorVar(\mathcal{C}_1)$ *and* $anchorVar(\mathcal{C}_2)$,
  then $M_{1t}(block) = M(block)$ *and from Lemma 8* $\ell \sqsubseteq M'_{2t}(block)$.
  From Lemma 7, $\ell \sqsubseteq M'_t(block)$.
  Because $\ell \sqsubseteq M(\underline{\mathcal{E}})$, *we have* $\ell \sqsubseteq M_1(pc).label$. *From Lemma 6,* $\ell \sqsubseteq M_{1t}(pc).label$.
  Due to (S-End), $\ell \sqsubseteq M_{1t}(pc).label \sqsubseteq M_t(block)$.
  So, $\ell \sqsubseteq M_t(block), M'_t(block)$.

- *If* $\neg anchorVar(\mathcal{C}_1)$ *and* $\neg anchorVar(\mathcal{C}_2)$,
  then $M_{1t}(block) = M(block)$ *and* $M'_{2t}(block) = M'(block)$.
  Due to (S-If) *and* (S-End), $M_t(block) = M_{1t}(block)$ *and* $M'_t(block) = M'_{2t}(block)$.
  So, if $\ell \not\sqsubseteq M_t(block)$, *then* $\ell \not\sqsubseteq M'_t(block)$ *and* $M_t(block) = M'_t(block)$.

Case $x, \underline{x}, \underline{\underline{x}}$***:*** *From Lemma 9, we get that for all* $x := \mathcal{E}'$ *in* $\mathcal{C}_1$ *and* $\mathcal{C}_2$, *we have* $\ell \sqsubseteq M_t(\underline{x}), M_t(\underline{\underline{x}}), M'_t(\underline{x}), M'_t(\underline{\underline{x}})$

Thus, we have $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$ *for all cases.*
Now, we want to show $consistent(M_t, M'_t, \ell)$.
From Lemma 6, $M_t(pc) = M(pc)$ *and* $M'_t(pc) = M'(pc)$.
If $M_t(pc).pop^n.top.d \wedge \ell \sqsubseteq M_t(pc).pop^n.label$, *then* $M(pc).pop^n.top.d \wedge \ell \sqsubseteq M(pc).pop^n.label$. *So,* $\ell \sqsubseteq M'(block)$ *or* $M'(pc).pop^n.top.d \wedge \ell \sqsubseteq M'(pc).pop^n.label$.
So, $\ell \sqsubseteq M'_t(block)$ *(from Lemma 7), or* $M'_t(pc).pop^n.top.d \wedge \ell \sqsubseteq M'_t(pc).pop^n.label$.
If $w \in M_t(pc).pop^n.top.V \wedge \ell \sqsubseteq M_t(pc).pop^n.label$, *then* $w \in M(pc).pop^n.top.V \wedge \ell \sqsubseteq M(pc).pop^n.label$.
Then $\ell \sqsubseteq M'(\underline{w})$ *and* $\ell \sqsubseteq M'(\underline{\underline{w}})$ *or* $w \in M'(pc).pop^n.top.V \wedge \ell \sqsubseteq M'(pc).pop^n.label$.
If $w \in M'(pc).pop^n.top.V \wedge \ell \sqsubseteq M'(pc).pop^n.label$, *then* $w \in M'_t(pc).pop^n.top.V \wedge \ell \sqsubseteq M'_t(pc).pop^n.label$.
If $\ell \sqsubseteq M'(\underline{w})$ *and* $\ell \sqsubseteq M'(\underline{\underline{w}})$ *and if there is no assignment* $w := \mathcal{E}'$ *to such a* w *in the if-statement, then* $\ell \sqsubseteq M'_t(\underline{w}) = M'(\underline{w})$ *and* $\ell \sqsubseteq M'_t(\underline{\underline{w}}) = M'(\underline{\underline{w}})$.
Otherwise, from Lemma 9, we get that $\ell \sqsubseteq M'_t(\underline{w}), M'_t(\underline{\underline{w}})$.
Thus, $consistent(M_t, M'_t, \ell)$. [$c1$]
We now prove [$c2$]. *Assume* Σ' *terminates normally and* $\ell \sqsubseteq M_t(block')$.
This means that Σ *is blocked at an assignment to anchor variable. So,* $anchorVar(\mathcal{C}_1)$ *holds, and thus, due to* (S-End), *we get* $\ell \sqsubseteq M'_2(pc) \sqsubseteq M'_t(block) = M'_{2t}(block) \sqcup M'_{2t}(pc)$. [$c2$]
Assume Σ' *terminates normally and* Σ *does not. Then* $\ell \sqsubseteq M_1(pc).label \sqsubseteq M_t(pc).label$.

Case 5. while $\mathcal{E}$ do $\mathcal{C}$ end

We prove $consistent(M_t, M'_t, \ell)$ *similarly to case* (S-If).
Induction on the maximum number of iterations in Σ *and* Σ'.
Base case: both Σ *and* Σ' *take* (S-Wl2).
Let $\ell \not\sqsubseteq M_t(pc).label$.
Then $\ell \not\sqsubseteq M(pc).label$ *and* $\ell \not\sqsubseteq M(\underline{\mathcal{E}})$,
because $M(pc).label, M(\underline{\mathcal{E}}) \sqsubseteq M_t(pc).label$.
So, $\ell \not\sqsubseteq M'(pc).label$, $\ell \not\sqsubseteq M'(\underline{\mathcal{E}})$, $M(pc) = M'(pc)$ *and* $M(\mathcal{E}) = M'(\mathcal{E})$.
Also, $\ell \not\sqsubseteq M(\underline{\underline{\mathcal{E}}})$ *and* $\ell \not\sqsubseteq M'(\underline{\underline{\mathcal{E}}})$, *because* $M(\underline{\underline{\mathcal{E}}}) \sqsubseteq M(\underline{\mathcal{E}})$ *and* $M'(\underline{\underline{\mathcal{E}}}) \sqsubseteq M'(\underline{\mathcal{E}})$.
Thus, $M(\underline{\mathcal{E}}) = M'(\underline{\mathcal{E}})$.
So, $M_t(pc) = M'_t(pc)$, *because* d *and* V *are the same for* Σ *and* Σ'.
Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. [$c1$], [$c3$]
Also, $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. [$c0$]
[$c2$] *is trivially true.*
Induction case.

Case 5.1. *Let* $\ell \not\sqsubseteq M(\underline{\mathcal{E}})$.
Then $\ell \not\sqsubseteq M'(\underline{\mathcal{E}})$ *and* $M(\mathcal{E}) = M'(\mathcal{E})$. *So, both* Σ *and* Σ' *take the same*

branch. If both take (S-WL2)*, then we follow the Base case. Assume that both take* (S-WL1)*.*

Case 5.1.1. *We have:*
$\Sigma = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1\rangle \xrightarrow{*}$
$\langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_2\rangle \xrightarrow{*} \langle \mathcal{C}_t^-,\ M_t\rangle$,
$\Sigma' = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M'\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1'\rangle \xrightarrow{*}$
$\langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_t^{-\prime},\ M_t'\rangle$.
Consider:
$\Sigma_1 = \langle \mathcal{C}; \textbf{end},\ M_1\rangle \xrightarrow{*} \langle \textbf{stop},\ M_2\rangle$,
$\Sigma_2 = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_2\rangle \xrightarrow{*} \langle \mathcal{C}_t,\ M_t\rangle$
$\Sigma_1' = \langle \mathcal{C}; \textbf{end},\ M_1'\rangle \xrightarrow{*} \langle \textbf{stop},\ M_2'\rangle$,
$\Sigma_2' = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_2'\rangle \xrightarrow{*} \langle \mathcal{C}_t',\ M_t'\rangle$.
Because $M(\underline{\mathcal{E}}) \sqsubseteq M(\underline{\mathcal{E}})$ *and* $M'(\underline{\mathcal{E}}) \sqsubseteq M'(\underline{\mathcal{E}})$*, we get* $\ell \not\sqsubseteq M(\underline{\mathcal{E}})$, $\ell \not\sqsubseteq M'(\underline{\mathcal{E}})$*, and thus,* $M(\underline{\mathcal{E}}) = M'(\underline{\mathcal{E}})$.
So, if $\ell \sqsubseteq M_1(pc).label$*, then* $\ell \sqsubseteq M(pc).label$*, and* $M(pc) = M'(pc)$ *and* $M_1(pc) = M_1'(pc)$.
Similarly, if $\ell \sqsubseteq M_1(pc).pop^n.label$*, with* $n \geq 1$*, then* $M_1(pc).pop^n = M_1'(pc).pop^n$.
Thus $M_1|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$.
From Lemma 4 and IH on $\mathcal{C}; \textbf{end}$, $\Sigma_1|_{\overline{\uparrow \ell}} = \Sigma_1'|_{\overline{\uparrow \ell}}$ *and* $M_2|_{\overline{\uparrow \ell}} = M_2'|_{\overline{\uparrow \ell}}$.
From IH on the max-number of iterations on Σ_2 *and* Σ_2', $\Sigma_2|_{\overline{\uparrow \ell}} = \Sigma_2'|_{\overline{\uparrow \ell}}$. *So,* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}}$. [c0]
If Σ *and* Σ' *terminate normally, then* Σ_2 *and* Σ_2' *terminate normally, and thus* $M_t|_{\overline{\uparrow \ell}} = M_t'|_{\overline{\uparrow \ell}}$. [c1]
Similarly, [c2] *and* [c3].

Case 5.1.2. *We have:*
$\Sigma = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1\rangle \xrightarrow{*}$
$\langle \mathcal{C}_{1t}^-; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_t\rangle$,
$\Sigma' = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M'\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1'\rangle \xrightarrow{*}$
$\langle \mathcal{C}_{1t}^{-\prime}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_t'\rangle$.
We follow the arguments above by using Lemma 4 and IH on $\mathcal{C}; \textbf{end}$.

Case 5.1.3. *We have:*
$\Sigma = \langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1\rangle$

$\xrightarrow{*} \langle \mathcal{C}_{1t}^{-}; \mathbf{end}; \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M_t \rangle$,
$\Sigma' = \langle \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M' \rangle$
$\rightarrow \langle \mathcal{C}; \mathbf{end}; \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M_1' \rangle$
$\xrightarrow{*} \langle \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M_2' \rangle \xrightarrow{*} \langle \mathcal{C}_t^{-\prime},\ M_t' \rangle$.
Consider:
$\Sigma_1 = \langle \mathcal{C}; \mathbf{end},\ M_1 \rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^{-},\ M_t \rangle$,
$\Sigma_1' = \langle \mathcal{C}; \mathbf{end},\ M_1' \rangle \xrightarrow{*} \langle \mathbf{stop},\ M_2' \rangle$,
$\Sigma_2' = \langle \mathbf{while}\ \mathcal{E}\ \mathbf{do}\ \mathcal{C}\ \mathbf{end},\ M_2' \rangle \xrightarrow{*} \langle \mathcal{C}_t^{-\prime},\ M_t' \rangle$.
As proved above, $M_1|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$.
From Lemma 4 and IH on $\mathcal{C}; \mathbf{end}$, *we get* $\Sigma_1|_{\overline{\uparrow \ell}} = \Sigma_1'|_{\overline{\uparrow \ell}}$.
$[c1]$ *is trivially true.*
Assume that Σ' *terminates normally and* $\ell \sqsubseteq M_t(block')$.
Then Σ_1' *terminates normally and* $\ell \sqsubseteq M_t(block')$.
Then $\ell \sqsubseteq M_2'(block)$. *From Lemma 7,* $\ell \sqsubseteq M_t'(block)$. $[c2]$
Similarly we prove $[c3]$.
For $[c0]$, *we want to show that* $\Sigma_2'|_{\overline{\uparrow \ell}} = \epsilon$. *We follow the same arguments as those in case* (S-SEQ).

Case 5.2. *Let* $\ell \sqsubseteq M(\underline{\mathcal{E}})$.
So, $\ell \sqsubseteq M'(\underline{\mathcal{E}})$.
Then Σ *and* Σ' *may take different branches.*
From Lemma 9, we get $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. $[c0]$
Assume that Σ *and* Σ' *terminate. From Lemma 9, for each* $x := \mathcal{E}$ *in the while-statement,* $\ell \sqsubseteq M_t(\underline{x}), M_t(\underline{\underline{x}}), M_t'(\underline{x}), M_t'(\underline{\underline{x}})$ *holds.*
Also, for each $r := ref(a)$ *in the while-statement we get* $\ell \sqsubseteq M_t(\underline{r}).r$ *and* $\ell \sqsubseteq M_t'(\underline{r}).r$.
From Lemma 6, $M_t(pc) = M(pc)$ *and* $M_t'(pc) = M'(pc)$.
Also, if $\ell \not\sqsubseteq M_t(block)$, *then there is no assignment to anchor variables in* $\mathcal{C}$ *and* $\ell \not\sqsubseteq M(block) = M_t(block)$. *So,* $M(block) = M'(block)$.
So, $M_t'(block) = M'(block) = M(block) = M_t(block)$.
Thus, $M_t|_{\overline{\uparrow \ell}} = M_t'|_{\overline{\uparrow \ell}}$. $[c1]$
If Σ *does not terminate, then* $anchorVar(\mathcal{C})$ *is true, and thus* $\ell \sqsubseteq M_t'(block)$.
$[c2]$
Also, $\ell \sqsubseteq M(\underline{\mathcal{E}}) \sqsubseteq M_t(pc).label$. *So* $[c3]$ *is trivially true.*

Case 6. end
$[c2]$ *is trivially true.*
Because $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$, $[c0]$ *holds.*
We prove $consistent(M_t, M_t', \ell)$.
Let $M_t(pc).pop^n.top.d$ *and* $\ell \sqsubseteq M_t(pc).pop^n.label$.

Then, $M(pc).pop^{n+1}.top.d$ *and* $\ell \sqsubseteq M(pc).pop^{n+1}.label$.
So, $\ell \sqsubseteq M'(block)$, *or* $M'(pc).pop^{n+1}.top.d$ *and* $\ell \sqsubseteq M'(pc).pop^{n+1}.label$.
Thus, $\ell \sqsubseteq M'_t(block)$ *(from Lemma 7), or* $M'_t(pc).pop^n.top.d$ *and* $\ell \sqsubseteq M'_t(pc).pop^n.label$.
Let $\ell \sqsubseteq M_t(pc).pop^n.label$.
Then $\ell \sqsubseteq M_t(pc).label$, *and thus* $\ell \sqsubseteq M(pc).label$.
So $\ell \sqsubseteq M'(pc).label$.
For a $w \in M_t(pc).pop^n.top.V$,
we have $w \in M(pc).pop^{n+1}.top.V$.
Also, $\ell \sqsubseteq M(pc).pop^{n+1}.label = M_t(pc).pop^n.label$.
So, $\ell \sqsubseteq M'(\underline{w})$ *and* $\ell \sqsubseteq M'(\underline{\underline{w}})$, *or* $w \in M'(pc).pop^{n+1}.top.V$ *and* $\ell \sqsubseteq M'(pc).pop^{n+1}.label$.
If $w \in M'(pc).pop^{n+1}.top.V$ *and* $\ell \sqsubseteq M'(pc).pop^{n+1}.label$, *then* $w \in M'_t(pc).pop^n.top.V$ *and* $\ell \sqsubseteq M'_t(pc).pop^n.label$.
If $\ell \sqsubseteq M'(\underline{w})$ *and* $\ell \sqsubseteq M'(\underline{\underline{w}})$ *and if* $w \in M'(pc).top.V$, *then* $\ell \sqsubseteq M'_t(\underline{w}), M'_t(\underline{\underline{w}})$, *otherwise* $\ell \sqsubseteq M'_t(\underline{w}) = M'(\underline{w})$ *and* $\ell \sqsubseteq M'_t(\underline{\underline{w}}) = M'(\underline{\underline{w}})$.
Thus, consistent(M_t, M'_t, ℓ).

Case 6.1. *Let* $\ell \sqsubseteq M_t(pc).label$.
Then $\ell \sqsubseteq M(pc).pop.label$. *So,* $\ell \sqsubseteq M'(pc).pop.label$, *due to* $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$.
So, $\ell \sqsubseteq M'_t(pc).label$.
Also, $\ell \sqsubseteq M(pc).label$ *and* $\ell \sqsubseteq M'(pc).label$.
Let $\ell \not\sqsubseteq M_t(pc).pop^n.label$.
Then, $\ell \not\sqsubseteq M(pc).pop^{n+1}.label$.
From hypothesis, $\ell \not\sqsubseteq M'(pc).pop^{n+1}.label$
and $M_t(pc).pop^n = M(pc).pop^{n+1} = M'(pc).pop^{n+1} = M'_t(pc).pop^n$.

Case 6.1.1. *Let* $M(pc).top.d$ *hold.*
So, $\ell \sqsubseteq M_t(block)$. *Because consistent*$(M, M', \ell)$, *we get* $\ell \sqsubseteq M'(block)$, *or* $M'(pc).top.d$ *holds.*
If $\ell \sqsubseteq M'(block)$, *then from Lemma 7,* $\ell \sqsubseteq M'_t(block)$.
If $M'(pc).top.d$ *holds, then* $\ell \sqsubseteq M'_t(block)$.

Case 6.1.2. *Let* $M(pc).top.d$ *be false.*
So, $M_t(block) = M(block)$.
If $\ell \not\sqsubseteq M_t(block)$, *then* $\ell \not\sqsubseteq M(block)$, *and thus* $\ell \sqsubseteq M'(block)$ *and* $M(block) = M'(block)$.
Assume for contradiction that $\ell \sqsubseteq M'_t(block)$. *This means that* $M'_t(block) \neq M'(block)$, *and so* $M'(pc).top.d$ *holds. From consistent*(M, M', ℓ), *we get* $\ell \sqsubseteq M(block)$
or $M(pc).top.d$ *is true, which are contradictions. So,* $\ell \not\sqsubseteq M'_t(block)$, *and* $M'_t(block) = M'(block) = M(block) = M_t(block)$.

Case 6.1.3. *Let $x \in M(pc).top.V$ and $x \in M'(pc).top.V$.*
Then $\ell \sqsubseteq M_t(\underline{x}), M_t(\underline{\underline{x}}), M'_t(\underline{x}), M'_t(\underline{\underline{x}})$.

Case 6.1.4. *Let $x \notin M(pc).top.V$ and $x \notin M'(pc).top.V$.*
Then the label and metalabel of x do not change in neither traces.

Case 6.1.5. *Let $x \in M(pc).top.V$ and $x \notin M'(pc).top.V$.*
From consistent(M, M', ℓ), we get $\ell \sqsubseteq M'(\underline{x}), M'(\underline{\underline{x}})$.
$\ell \sqsubseteq M_t(\underline{x}), M_t(\underline{\underline{x}}), M'_t(\underline{x}) = M'(\underline{x}), M'(\underline{\underline{x}}) = M'_t(\underline{\underline{x}})$.

Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. [c1], [c3]

Case 6.2. *Let $\ell \not\sqsubseteq M_t(pc).label$.*
Then $\ell \not\sqsubseteq M(pc).pop.label$. So, $\ell \not\sqsubseteq M'(pc).pop.label$ and $M(pc).pop.label = M'(pc).pop.label$.
Thus $\ell \not\sqsubseteq M'_t(pc).label$ and $M_t(pc) = M'_t(pc)$.
If $\ell \sqsubseteq M(pc).label$, then we work as above.
Let $\ell \not\sqsubseteq M(pc).label$.
Then $\ell \not\sqsubseteq M'(pc).label$ and $M(pc) = M'(pc)$.
So, $M(pc).top.d = M'(pc).top.d$ and $M(pc).top.V = M'(pc).top.V$.
For an $x \in V$, if $\ell \not\sqsubseteq M_t(\underline{x})$, then $\ell \not\sqsubseteq M(\underline{x})$, because $M_t(\underline{x}) = M(\underline{x}) \sqcup M(pc).label$, and thus $\ell \not\sqsubseteq M'(\underline{x})$. So, $\ell \not\sqsubseteq M(\underline{\underline{x}})$ and $\ell \not\sqsubseteq M'(\underline{\underline{x}})$, and thus, $M(\underline{x}) = M'(\underline{x})$ and $M(\underline{\underline{x}}) = M'(\underline{\underline{x}})$. Because $M(pc).label = M'(pc).label$, we get $M_t(\underline{x}) = M'_t(\underline{x})$ and $M_t(\underline{\underline{x}}) = M'_t(\underline{\underline{x}})$.

Case 6.2.1. *Let $\ell \not\sqsubseteq M_t(block)$.*
Then $\ell \not\sqsubseteq M(block)$ (independently to $M(pc).top, d$).
So, $\ell \not\sqsubseteq M'(block)$ and $M(block) = M'(block)$.
Because $M(pc) = M'(pc)$ and $M(pc).top.d = M'(pc).top.d$, we have $M_t(block) = M'_t(block)$ and $\ell \not\sqsubseteq M'_t(block)$.

Case 6.2.2. *If $\ell \sqsubseteq M_t(block)$, then $\ell \sqsubseteq M'_t(block)$.*

Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. [c1], [c3].

□

Lemma 4. $\langle \mathcal{C}_0^-,\ M_0 \rangle \xrightarrow{*} \langle \mathcal{C}^-,\ M \rangle \wedge healthy(M_0) \Rightarrow healthy(M)$.

Proof. By induction on the OS rules.
We use Lemma 5 to prove the case of (S-Asgn2). □

Lemma 5. $\forall x. M(\underline{x}) \sqsubseteq M(\underline{x}) \Rightarrow M(\underline{\mathcal{E}}) \sqsubseteq M(\underline{\mathcal{E}})$.

Proof. By induction on the structure of $\mathcal{E}$. □

Lemma 6. *If* $\langle \mathcal{C},\ M\rangle \xrightarrow{*} \langle \textbf{stop},\ M_t\rangle$, *then* $M(pc) = M_t(pc)$.

Proof. By induction on the OS rules. □

Lemma 7. *If* $\langle \mathcal{C},\ M\rangle \xrightarrow{*} \langle \mathcal{C}^-,\ M'\rangle$, *then* $M(block) \sqsubseteq M'(block)$.

Proof. By induction on the OS rules. □

Lemma 8. *If* $\langle \mathcal{C}^-,\ M\rangle \xrightarrow{*} \langle \textbf{stop},\ M_t\rangle$ *and* $\ell \sqsubseteq M(pc).label$ *and* $anchorVar(\mathcal{C}^-)$, *then* $\ell \sqsubseteq M_t(block)$. *Additionally, if* $\mathcal{C}^-$ *is a conditional command with guard* $\mathcal{E}$, *and* $\ell \sqsubseteq M(\underline{\mathcal{E}})$ *and* $anchorVar(\mathcal{C}^-)$, *then* $\ell \sqsubseteq M_t(block)$.

Proof. Induction on the OS rules.

Case 1. (S-Asgn1)
$\ell \sqsubseteq M(block) \sqcup M(\underline{\mathcal{E}}) \sqcup M(pc).label = M_t(block)$.

Case 2. (S-Asgn2)
Trivially true.

Case 3. (S-Asgn3)
Trivially true.

Case 4. (S-Asgn4)
$\ell \sqsubseteq M(block) \sqcup M(\underline{\mathcal{E}}) \sqcup M(pc).label \sqcup M(\underline{r}).r = M_t(block)$.

Case 5. (S-Seq)
We have $\langle \mathcal{C}_1;\mathcal{C}_2,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_2,\ M_1\rangle \xrightarrow{*} \langle \textbf{stop},\ M_t\rangle$.
From IH on $\mathcal{C}_1$ *and if* $anchorVar(\mathcal{C}_1)$, *we get* $\ell \sqsubseteq M_1(block)$. *From Lemma 7,* $\ell \sqsubseteq M_t(block)$.
If $anchorVar(\mathcal{C}_2)$, *from Lemma 6 that gives* $M_1(pc) = M(pc)$, *and from Lemma 4 and IH on* $\mathcal{C}_2$, *we get* $\ell \sqsubseteq M_t(block)$.

Case 6. (S-If)
We have:
$\langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M\rangle \rightarrow \langle \mathcal{C}_1;\textbf{end},\ M_1\rangle \xrightarrow{*} \langle \textbf{end},\ M_e\rangle \rightarrow \langle \textbf{stop},\ M_t\rangle$.
We have $\ell \sqsubseteq M_1(pc).label$.
From IH on $\mathcal{C}_1$, *if* $anchorVar(\mathcal{C}_1)$, *then we get* $\ell \sqsubseteq M_e(block)$.
From Lemma 7, $\ell \sqsubseteq M_t(block)$.
If $anchorVar(\mathcal{C}_2)$, *then* $M_1(pc).top.d = true$.
From Lemma 6, $M_e(pc) = M_1(pc)$, *and thus* $M_e(pc).top.d = true$.
So, $\ell \sqsubseteq M_e(block) \sqcup M_1(pc).label = M_e(block) \sqcup M_e(pc).label = M_t(block)$.

Case 7. (S-WL1)
From Lemma 4 and IH on $\mathcal{C}$ and Lemma 7.

Case 8. (S-WL2)
We have $\langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M\rangle \rightarrow \langle \textbf{end},\ M_e\rangle \rightarrow \langle \textbf{stop},\ M_t\rangle$.
If $anchorVar(\mathcal{C})$, *then* $\ell \sqsubseteq M(pc).label \sqsubseteq M_e(pc).label \sqsubseteq M_e(block) \sqcup M_e(pc).label = M_t(block)$.

Case 9. (S-END)
Trivially true.

Now we concentrate on while-statements and if-statements. Consider
$\langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M\rangle \rightarrow$
$\langle \mathcal{C}_1; \textbf{end},\ M_1\rangle \xrightarrow{*} \langle \textbf{end},\ M_e\rangle \rightarrow \langle \textbf{stop},\ M_t\rangle$.
We have $\ell \sqsubseteq M(\underline{\mathcal{E}}) \sqsubseteq M_1(pc).label$.
So, we apply the above result to $\langle \mathcal{C}_1; \textbf{end},\ M_1\rangle$ and get $\ell \sqsubseteq M_t(block)$.
Consider $\langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M\rangle \rightarrow$
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1\rangle \xrightarrow{*}$
$\langle \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_2\rangle \xrightarrow{*} \langle \textbf{stop},\ M_t\rangle$.
We have $\ell \sqsubseteq M(\underline{\mathcal{E}}) \sqsubseteq M_1(pc).label$.
So, we apply the above result to
$\langle \mathcal{C}; \textbf{end}; \textbf{while } \mathcal{E} \textbf{ do } \mathcal{C} \textbf{ end},\ M_1\rangle$ and get $\ell \sqsubseteq M_t(block)$.

□

Lemma 9. *If* $\Sigma = \langle \mathcal{C}_i^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_j^-,\ M_t\rangle$ *and* $\mathcal{C}_i^-$ *is a conditional command with guard* $\mathcal{E}$, *and* $\ell \sqsubseteq M(\underline{\mathcal{E}})$, *then*

- $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$,
- *if* Σ *terminates and* $w := \mathcal{E}' \in \mathcal{C}_i^-$ *then* $\ell \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$, *and*
- *if* Σ *terminates and* $r := ref(a) \in \mathcal{C}_i^-$ *then* $\ell \sqsubseteq M_t(\underline{r}).r$.

Proof. First we prove:
if $\Sigma = \langle \mathcal{C}_i^-,\ M\rangle \xrightarrow{*} \langle \mathcal{C}_j^-,\ M_t\rangle$ and $\ell \sqsubseteq M(pc).label$, and $\ell \sqsubseteq M(\underline{\mathcal{E}})$, then

- $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$,
- if Σ terminates and $w := \mathcal{E}' \in \mathcal{C}_i^-$ then $\ell \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$, and
- if Σ terminates and $r := ref(a) \in \mathcal{C}_i^-$ then $\ell \sqsubseteq M_t(\underline{r}).r$.

We use induction on $\mathcal{C}$.

Case 1. (S-Asgn1)
The assignment is executed only if $\ell \sqsubseteq M(pc).label \sqsubseteq M(\underline{a})$. *So,* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.

Case 2. (S-Asgn2)
$\ell \sqsubseteq M(pc).label \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$. *So,* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.

Case 3. (S-Asgn3)
$\ell \sqsubseteq M(pc).label \sqsubseteq M_t(\underline{r}).r$. *So,* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.

Case 4. (S-Asgn4)
$\ell \sqsubseteq M(pc).label \sqsubseteq M_t(\underline{r}).t$. *So,* $\ell \sqsubseteq M(\underline{M(r)})$, *and thus* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.

Case 5. (S-Seq)
If Σ *is blocked in* $\mathcal{C}_1^-$, *then we apply IH on* $\mathcal{C}_1^-$ *and get* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.
If Σ *executes* $\mathcal{C}_1^-$, *from IH on* $\mathcal{C}_1^-$, *we get* $w := \mathcal{E}' \in \mathcal{C}_1^- \Rightarrow \ell \sqsubseteq M_1(\underline{w}), M_1(\underline{\underline{w}})$ *and* $\Sigma_1|_{\overline{\uparrow \ell}} = \epsilon$.
From Lemma 6, $M_1(pc) = M(pc)$.
If Σ *is blocked in* $\mathcal{C}_2^-$, *we can use Lemma 4 and apply IH on* $\mathcal{C}_2^-$ *and get* $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$.
If $\mathcal{C}_2^-$ *terminates, then* $w := \mathcal{E}' \in \mathcal{C}_2^- \Rightarrow \ell \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$.
So, if $w := \mathcal{E}' \in \mathcal{C}_1^-; \mathcal{C}_2^-$, *then*
$w := \mathcal{E}' \in \mathcal{C}_2^- \Rightarrow \ell \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$
$w := \mathcal{E}' \notin \mathcal{C}_2^- \wedge w := \mathcal{E}' \in \mathcal{C}_1^- \Rightarrow \ell \sqsubseteq M_1(\underline{w}) = M_t(\underline{w}) \wedge \ell \sqsubseteq M_1(\underline{\underline{w}}) = M_t(\underline{\underline{w}})$.

Case 6. (S-If)
We have:
$\Sigma \leq \langle \textbf{if } \mathcal{E} \textbf{ then } \mathcal{C}_1 \textbf{ else } \mathcal{C}_2 \textbf{ end},\ M\rangle \rightarrow \langle \mathcal{C}_1; \textbf{end},\ M_1\rangle \overset{*}{\rightarrow} \langle \textbf{end},\ M_e\rangle \rightarrow \langle \textbf{stop},\ M_t\rangle$.
We have $\ell \sqsubseteq M(pc).label \sqsubseteq M_1(pc).label$.
From IH on $\mathcal{C}_1$, $\Sigma|_{\overline{\uparrow \ell}} = \epsilon$, *and* $w := \mathcal{E}' \in \mathcal{C}_1 \Rightarrow \ell \sqsubseteq M_e(\underline{w}), M_e(\underline{\underline{w}})$.
But, $w := \mathcal{E}' \in \mathcal{C}_2 \Rightarrow \ell \sqsubseteq M_1(pc).label = M_e(pc).label \sqsubseteq M_t(\underline{w}), M_t(\underline{\underline{w}})$.
Also $w := \mathcal{E}' \in \mathcal{C}_1 \Rightarrow M_e(\underline{w}) \sqsubseteq M_t(\underline{w})$ *and* $M_e(\underline{\underline{w}}) \sqsubseteq M_t(\underline{\underline{w}})$.

Case 7. (S-Wl)
Similarly to above.

Case 8. (S-End)
Trivially true.

From the statement proved and from $\ell \sqsubseteq M(\underline{\mathcal{E}})$, we get what we want. □

We prove that when substituting (S-Asgn1) and (S-Asgn2) with (S'-Asgn1) and (S'-Asgn2), correspondingly, the resulting operational semantics satisfy Theorem 1 for H and L labels.
First, we prove that Theorem 1 holds for H and L labels when we use rule (S'-Asgn1) instead of (S-Asgn1). Then, given that the new operational semantics, where (S-Asgn1) is substituted by (S'-Asgn1) does not use metalabels, but they only update metalabels, we deduce that if metalabels are removed from the operational semantics, Theorem 1 still holds for H and L labels. In particular, (S-Asgn2) can be substituted by (S'-Asgn2) and the Theorem will still hold.

Proposition 10. *Assuming rule* (S-Asgn1) *is substituted by rule* (S'-Asgn1)*, and considering a two level lattice* $\langle\{\mathtt{H},\mathtt{L}\},\ \sqsubseteq\rangle$*, with* $\mathtt{L}=\bot$ *and* $\mathtt{L}\sqsubseteq\mathtt{H}$*, then Theorem 1 holds for* $\ell=\mathtt{H}$*.*

Proof. We follow the strategy used to prove Case 1 in the proof of Theorem 1. Instead of writing $\ell\not\sqsubseteq\ell'$ for a label ℓ', we may write $\ell'=\mathtt{L}$, because $\ell=\mathtt{L}$. And, instead of writing $\ell\sqsubseteq\ell'$, we may write $\ell'=\mathtt{H}$.

We first prove the second part of c1; afterwards, we prove c0-c3 by cases.
If both Σ and Σ' terminate, $M_t(pc)=M(pc)$, $M'_t(pc)=M'(pc)$, $M_t(block)=M(block)$, $M'_t(block)=M'(block)$, $\forall w.M_t(\underline{w})=M(\underline{w})$, $\forall w.M'_t(\underline{w})=M'(\underline{w})$, $\forall w.M_t(\underline{\underline{w}})=M(\underline{\underline{w}})$, and $\forall w.M'_t(\underline{\underline{w}})=M'(\underline{\underline{w}})$.
Because $\mathit{consistent}(M,M',\ell)$, from the above we also get $\mathit{consistent}(M_t,M'_t,\ell)$.
We also have $M(\underline{a})=M'(\underline{a})$: by definition, $\ell\not\sqsubseteq M(\underline{a})=M'(\underline{a})=\bot$, which implies $\underline{a}\in\mathrm{dom}(M|_{\overline{\uparrow\ell}})$, and because $M|_{\overline{\uparrow\ell}}=M'|_{\overline{\uparrow\ell}}$, we have $M(\underline{a})=M'(\underline{a})$.

Case 1. *Let* $M(\underline{a})=M'(\underline{a})=\mathtt{L}$*. We first prove that the command is executed normally in both memories or blocked in both memories. W.l.o.g., assume that the command is executed normally in* M*. That is,* $M(\underline{\mathcal{E}})\sqcup M(pc).label\sqcup M(block')\sqsubseteq M(\underline{a})$*, where* $M(block')=M(block)\sqcup M(pc).label$*. This implies that all members of the supremum are* L*, that is,* $M(\underline{\mathcal{E}})=M(pc).label=M(block')=\mathtt{L}$*; thus,* $M(block)=\mathtt{L}$*. Hence the labels involved in these expressions are in* $M|_{\overline{\uparrow\ell}}$*.*
Then, because $M|_{\overline{\uparrow\ell}}=M'|_{\overline{\uparrow\ell}}$*, they are also in* $M'|_{\overline{\uparrow\ell}}$ *and the two memories agree on them, that is,* $M(\mathcal{E})=M'(\mathcal{E})$*,* $M(pc)=M'(pc)$*,* $M(block)=M'(block)=\mathtt{L}$*, and* $M(\underline{\mathcal{E}})=M'(\underline{\mathcal{E}})=\mathtt{L}$*.*
$M(\underline{a})=M'(\underline{a})$ *by the argument above. Thus,* $M'(\underline{\mathcal{E}})\sqcup M'(pc).label\sqcup M'(block')\sqsubseteq M'(\underline{a})$*, as wanted.*

Case 1.1. *The command is executed normally in both memories. Then* $\Sigma=\langle a:=\mathcal{E},\ M\rangle\rightarrow\langle\mathbf{stop},\ M[a\mapsto M(\mathcal{E}),block\mapsto M(block')]\rangle$*, and* $\Sigma'=\langle a:=\mathcal{E},\ M'\rangle\rightarrow\langle\mathbf{stop},\ M'[a\mapsto M'(\mathcal{E}),block\mapsto M'(block')]\rangle$*.*

We have $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \langle a,\ M(\mathcal{E})\rangle$ *because* $M(\mathcal{E}) = M'(\mathcal{E})$. *Thus,* $[c0]$ *holds.*
Also, we have $M_t(a) = M'_t(a)$ *and* $M_t(block) = M'_t(block)$. *Hence* $[c1], [c3]$.
Finally, $[c2]$ *is trivially true.*

Case 1.2. *Both traces are blocked. Then* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$, *blocked.* $[c0]$.
Finally, $[c1], [c2], [c3]$ *are trivially true.*

Case 2. *Let* $M(\underline{a}) = M'(\underline{a}) = \mathtt{H}$.
Then $M_t(\underline{a}) \sqcup M_t(block) = \mathtt{H}$, *and hence, by Lemma 2, we have* $\Sigma|_{\overline{\uparrow \ell}} = \Sigma'|_{\overline{\uparrow \ell}} = \epsilon$. $[c0]$
Because $M(\underline{a}) = M_t(\underline{a}) = \mathtt{H}$, *we have* $a \notin M_t|_{\overline{\uparrow \ell}}$, *and hence, we only have to check variable block to prove* $[c1]$.

Case 2.1. *Let* $\mathtt{L} = M_t(block) = M(block) \sqcup M(\underline{\underline{\mathcal{E}}}) \sqcup M(pc).label$ *and both* Σ, Σ' *terminate.*
Then, $M(block) = \mathtt{L}$ *and* $M(pc).label = \mathtt{L}$.
Because $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$, $M'(block) = M'(pc).label = \mathtt{L}$ *holds.*
Also, $M(pc) = M'(pc)$.
So, $M'_t(block) = M_t(block) = \mathtt{L}$.
Thus, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$. $[c1], [c3]$
$[c2]$ *is trivially true.*

Case 2.2. *Let* $M_t(block) = \mathtt{H}$ *and both* Σ, Σ' *terminate.*
By symmetry of the preceding case, $M'_t(block) = \mathtt{H}$.
So, $M_t|_{\overline{\uparrow \ell}} = M'_t|_{\overline{\uparrow \ell}}$, *since block is not in their domain.* $[c1], [c3]$
$[c2]$ *is trivially true.*

Case 2.3. *Let* Σ' *terminate and* Σ *do not.*
$[c1]$ *is trivially true. We have:*
$M'_t(block) = block' = M'(block) \sqcup M'(pc).label$ *and*
$M_t(block') = M(block) \sqcup M(pc).label$.
$M'_t(block) = \mathtt{L} \Rightarrow$
$M'(pc).label = M'(block) = \mathtt{L} \Rightarrow$ *(because* $M|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}}$*)*
$M(pc).label = M(block) = \mathtt{L}$, *and*
$M(pc) = M'(pc)$, *and* $M(block) = M'(block) = M_t(block)$.
So $M_t(block) = \mathtt{L}$, *as wanted. Thus,* $M'_t(block) = \mathtt{H} \Rightarrow M_t(block') = \mathtt{H}$. $[c2]$
We have $\langle a := \mathcal{E},\ M'\rangle \leq \Sigma'$, *so* $M''_t = M'$, *and* $M''_t|_{\overline{\uparrow \ell}} = M'|_{\overline{\uparrow \ell}} = M|_{\overline{\uparrow \ell}} = M_t|_{\overline{\uparrow \ell}}$. $[c3]$

We also need to change parts of the proof of Theorem 1 that use (S-Asgn1). Also, parts of the proofs of Lemmata that use (S-Asgn1).
In Theorem 1, these parts are found in (S-Seq) and (S-Wl), concerning

the proof of $\Sigma_2'|_{\overline{\uparrow \ell}} = \epsilon$.
Case 3.3.2 ($M_t(pc).label = \texttt{L}$) becomes as follows. From IH[$c3$] on $\mathcal{C}_1^-$, Σ_1, Σ_1', there exists $\langle \mathcal{C}_1^-,\ M' \rangle \xrightarrow{*} \langle \mathcal{C}_{1t}^-,\ M_1' \rangle \leq \Sigma_1'$ and $M_t|_{\overline{\uparrow \ell}} = M_1'|_{\overline{\uparrow \ell}}$.
So, $\ell \not\sqsubseteq M_1'(pc).label$ and $M_t(pc) = M_1'(pc)$.
Because Σ_1 was blocked and Σ_1' was not, the values of $\underline{\mathcal{E}}$, and *block* may be different at these traces. Because $M_t|_{\overline{\uparrow \ell}} = M_{1't}|_{\overline{\uparrow \ell}}$, and because $\underline{\mathcal{E}}$ and *block* can either be H or L, we have that $M_t(\underline{\mathcal{E}}) = M_1'(\underline{\mathcal{E}})$ and $M_t(block) = M_1'(block')$. So, it cannot be the case that Σ_1 is blocked and Σ_1' is not blocked. So, it cannot be the case that $M_t(pc).label = \texttt{L}$. Thus, Case 3.3.2 cannot happen when labels H and L are used. We follow the same arguments for Case 5.1.3. □