Nontransitive Security Types for Coarse-grained Information Flow Control

Yi Lu
School of Computer Science
Queensland University of Technology
Brisbane, Australia
yt.lu@qut.edu.au

Chenyi Zhang
College of Information Science and Technology
Jinan University
Guangzhou, China
chenyi_zhang@jnu.edu.cn

***Abstract*—Language-based information flow control (IFC) aims to provide guarantees about information propagation in computer systems having multiple security levels. Existing IFC systems extend the lattice model of Denning's, enforcing transitive security policies by tracking information flows along with a partially ordered set of security levels. They yield a transitive noninterference property of either confidentiality or integrity. In this paper, we explore IFC for security policies that are not necessarily transitive. Such nontransitive security policies avoid unwanted or unexpected information flows implied by transitive policies and naturally accommodate high-level coarse-grained security requirements in modern component-based software.**

We present a novel security type system for enforcing nontransitive security policies. Unlike traditional security type systems that verify information propagation by subtyping security levels of a transitive policy, our type system relaxes strong transitivity by inferring information flow history through security levels and ensuring that they respect the nontransitive policy in effect. Such a type system yields a new nontransitive noninterference property that offers more flexible information flow relations induced by security policies that do not have to be transitive, therefore generalizing the conventional transitive noninterference. This enables us to directly reason about the extent of information flows in the program and restrict interactions between security-sensitive and untrusted components.

***Index Terms*—Language-based security, information flow control, noninterference, security types, type systems**

I. Introduction

Information flow control (IFC) is the problem of ensuring secure information flow according to specified policies within computer systems having multiple security levels. The standard information flow security property, noninterference [1], guarantees confidentiality that public outputs of a program are not influenced by private inputs, or vice versa for integrity. In a programming language setting, language-based techniques, such as static type systems [2], [3] or dynamic runtime monitoring [4], [5], are used for tracking and enforcing information flow policies. Our focus in this paper is on IFC enforced through type systems.

Most language-based IFC systems extend the lattice model of Denning [6], [7] to deal with transitive policies, which assume security levels to be partially ordered in a security lattice. Such partially ordered security levels are used to abstract program values (inputs, outputs and intermediate values in variables) and to track dependencies between them. While transitive policies can be verified through the simple subtyping of security levels associated with program variables [2], nontransitive policies are more flexible especially in expressing coarse-grained security requirements of interactions among software components.

Modern applications are composed of reusable software components from different sources. Untrusted code (e.g. components downloaded from the Internet) may be executed in the same process, alongside sensitive system code. Instead of trusting developer-provided security policies, users (such as application developers, builders or deployers that use the downloaded components to build applications) must specify security policies to protect sensitive system components from untrusted code. Without knowing implementation details, the users can express only *coarse-grained* policies on downloaded components according to their expected functionalities, where the entire code of a component is associated with a single security level (i.e. not labeling all variables individually). For example, the Java programming language provides a sandboxing mechanism that makes the use of user-provided security policy files to assign privileges to code bases [8]. Furthermore, systems with coarse-grained policies are easier to design and implement, reducing the security annotation burden that fine-grained systems impose on developers [9], [10].

In a transitive IFC system, the information in a security level can flow to all security levels by the transitive closure of the information flow relation defined by its policy. This makes it difficult to use transitive policies to express coarse-grained security requirements. For example, a component Alice may trust only another component Bob with her information, however due to implied transitive relations, her information may flow not only to Bob but also indirectly to all components that Bob trusts, which is undesirable for Alice. Furthermore, mutual and circular information flows among security levels are precluded, for instance, Alice and Bob cannot send information to each other unless both components are in the same security level (i.e. they must share the same information flow relations with any other component).

In this paper, we explore IFC for security policies that are not necessarily transitive, in order to avoid unwanted flow

Yi Lu and Chenyi Zhang are the co-first authors. Chenyi Zhang is the corresponding author.

DOI 10.1109/CSF49147.2020.00022

relations implied by transitivity and support more flexible flow structures than transitive closures in traditional IFC systems. (Note, our notion of nontransitive noninterference is not to be confused with the previous notion of *intransitive noninterference*, as discussed in Section III.) Such nontransitive policies can facilitate reasoning about and accommodate coarse-grained security requirements. For example, it is safe for Bob to send his own information to Charlie (who is untrusted by Alice), as long as Bob does not expose Alice's information to Charlie. Nontransitive security policies can easily express permitted information flow relations from Alice to Bob as well as from Bob to Charlie, while still forbidding undesired information flow relation from Alice to Charlie (because there is no implied relation from Alice to Charlie in the absence of transitivity). Moreover, mutual and circular information flow relations across components in different security levels can be supported. For example, Alice and Bob may trust each other without having to share the same trust to Charlie, by specifying flow relations from Alice to Bob, from Bob to Alice, and from Bob to Charlie (but not from Alice to Charlie). Nontransitive policies generalize transitive policies by requiring that transitive information flow relationships must be specified explicitly rather than derivable implicitly. Therefore, transitive security models are a special case of nontransitive models where all transitive closures are explicitly specified.

Nontransitive security models are particularly useful in modern component-based software development, where high-level coarse-grained information flow policies are specified by the users or deployers of the code bases, components or modules, who do not have to trust the developers to provide low-level fine-grained security policies on their implementation. Let's consider a simple program in Fig. 1 that is composed from three code components, named `Alice`, `Bob` and `Charlie`, developed by different developers. Each component may enclose some data and operations, like usual classes or modules. Let's assume that `Alice` would share her information with `Bob` (e.g. for performing certain functionality), but `Charlie` must never receive Alice's information. There are two operations in `Bob`; `Bob.bad` should be rejected because it sends `Alice`'s data to `Charlie`, while `Bob.good` should be allowed because `Bob` only sends his own data to `Charlie` without revealing `Alice`'s. We will examine both transitive and nontransitive IFC systems in specifying the desired security policies to meet these security requirements.

A transitive IFC system with fine-grained analysis requires annotating (or inferring) the label of every intermediate value, and then carefully tracks dependencies among these values. In the example shown in Fig. 1, four security levels ($\mathbb{L}_{\mathtt{A}}$, $\mathbb{L}_{\mathtt{B1}}$, $\mathbb{L}_{\mathtt{B2}}$, $\mathbb{L}_{\mathtt{C}}$), partially ordered by the transitive relation $\geq$, are used to label program variables in the components:

$$\begin{aligned} \texttt{Alice.data} &: \mathbb{L}_{\mathtt{A}} \\ \texttt{Bob.data1} &: \mathbb{L}_{\mathtt{B1}} \\ \texttt{Bob.data2} &: \mathbb{L}_{\mathtt{B2}} \\ \texttt{Charlie.data} &: \mathbb{L}_{\mathtt{C}} \end{aligned}$$

```
Alice {
  data;
  main() {
    Bob.receive(data);
    Bob.good();
    Bob.bad();
  }
}

Bob {
  data1;
  data2;
  receive(x) { data1 = x; }
  good() { Charlie.receive(data2); }
  bad() { Charlie.receive(data1); }
}

Charlie {
  data;
  receive(x) { data = x; }
}
```

Fig. 1. Charlie must not receive Alice's information.

A transitive policy is specified as a couple of (permitted) information flow relations:

$$\begin{aligned} \mathbb{L}_{\mathtt{A}} &\geq \mathbb{L}_{\mathtt{B1}} \\ \mathbb{L}_{\mathtt{B2}} &\geq \mathbb{L}_{\mathtt{C}} \end{aligned}$$

To correctly track dependencies between program values, variables in the same component `Bob` are separated into two security levels $\mathbb{L}_{\mathtt{B1}}$ and $\mathbb{L}_{\mathtt{B2}}$. This allows the IFC system to distinguish the data received from `Alice` (stored in the variable `Bob.data1`) and the data sent to `Charlie` (loaded from the variable `Bob.data2`), and to verify that values labeled security level $\mathbb{L}_{\mathtt{A}}$ can never flow to variables labeled $\mathbb{L}_{\mathtt{C}}$. The operation `Bob.good` is permitted because `Bob.data2` is labeled $\mathbb{L}_{\mathtt{B2}}$ which is allowed to flow to $\mathbb{L}_{\mathtt{C}}$ by the policy in effect. Another operation `Bob.bad` is rejected by checking the labels because `Bob.data1` is labeled $\mathbb{L}_{\mathtt{B1}}$ and there is no derivable flow relation from $\mathbb{L}_{\mathtt{B1}}$ to $\mathbb{L}_{\mathtt{C}}$ by the policy.

Such low-level labeling and policies at the granularity of program variables are typically specified by the developers who have the domain knowledge of the software components. However, developers of code downloaded from the Internet may not necessarily be trusted; such code may be malicious or contain developer-induced security issues. Therefore, it would be desirable that the users of untrusted components are able to specify high-level security policies at the granularity of components such as modules and code bases, without having to know their implementation details.

The developers of software components generally cannot anticipate how their components may be used in all possible application contexts which have different application-specific information flow requirements. For example, the developer of `Bob` would not specify separate security levels for `Bob.data1` and `Bob.data2` unless he knew that the component would

be used in only applications whose security policies forbid information flow from `Alice` to `Charlie`. On the other hand, user-provided security policies can specify application-specific information flow requirements, facilitating the reusability of software components.

Next, we try to use information flow policies coarsely, by associating a single security level with an entire component and not labeling any variable individually. This enables the user of the components to label each component based on their intended functionalities, rather than their implementation details (i.e. `Bob.data1` and `Bob.data2`):

$$\texttt{Alice} : \mathbb{L}_\texttt{A}$$
$$\texttt{Bob} : \mathbb{L}_\texttt{B}$$
$$\texttt{Charlie} : \mathbb{L}_\texttt{C}$$

Now, all variables within the scope of a component implicitly share the same label on the component. For example, both `Bob.data1` and `Bob.data2` are labeled by the same security level $\mathbb{L}_\texttt{B}$. Since the IFC system cannot distinguish these variables from their security levels, the operations `Bob.good` and `Bob.bad` would either pass together or fail together by checking their labels. It is hard to provide a meaningful transitive policy at component-level to capture the security requirement of the application. For instance, the user may specify a transitive policy:

$$\mathbb{L}_\texttt{A} \geq \mathbb{L}_\texttt{B}$$
$$\mathbb{L}_\texttt{B} \geq \mathbb{L}_\texttt{C}$$

By transitivity, the policy implicitly provides $\mathbb{L}_\texttt{A} \geq \mathbb{L}_\texttt{C}$; therefore, both operations `Bob.good` and `Bob.bad` would be permitted by the IFC system. In other cases, if the user specifies only $\mathbb{L}_\texttt{A} \geq \mathbb{L}_\texttt{B}$, then both `Bob.good` and `Bob.bad` are rejected by the IFC system as there is no allowed information flow from $\mathbb{L}_\texttt{B}$ to $\mathbb{L}_\texttt{C}$. Conversely, if only $\mathbb{L}_\texttt{B} \geq \mathbb{L}_\texttt{C}$ is specified then both operation are accepted (`Alice.main` is rejected for no permitted information flow from $\mathbb{L}_\texttt{A}$ to $\mathbb{L}_\texttt{B}$).

Finally, we employ a nontransitive IFC system for the user to label and specify coarse-grained information flow policies, such that all variables within a component share the same label. Significantly, these security levels ($\mathbb{S}_\texttt{A}$, $\mathbb{S}_\texttt{B}$, $\mathbb{S}_\texttt{C}$) are not necessarily partially ordered by a lattice.

$$\texttt{Alice} : \mathbb{S}_\texttt{A}$$
$$\texttt{Bob} : \mathbb{S}_\texttt{B}$$
$$\texttt{Charlie} : \mathbb{S}_\texttt{C}$$

A nontransitive security policy provides an arbitrary information flow relation $\trianglerighteq$ to define permissible information flows (in general, this relation would be reflective to be useful). For example, $\mathbb{S}_\texttt{A} \trianglerighteq \mathbb{S}_\texttt{B}$ specifies that information can flow from security level $\mathbb{S}_\texttt{A}$ to another security level $\mathbb{S}_\texttt{B}$.

$$\mathbb{S}_\texttt{A} \trianglerighteq \mathbb{S}_\texttt{B}$$
$$\mathbb{S}_\texttt{B} \trianglerighteq \mathbb{S}_\texttt{C}$$

In the absence of transitivity, information flow relations cannot be derived but must be explicitly specified in the policy. With such a policy, information associated with $\mathbb{S}_\texttt{A}$ must not flow to components associated with $\mathbb{S}_\texttt{C}$ because there is no derivable information flow relation from $\mathbb{S}_\texttt{A}$ to $\mathbb{S}_\texttt{C}$. In our example, this policy means there may be information flows from `Alice` to `Bob` and from `Bob` to `Charlie` but `Alice`'s information must never flow to `Charlie` directly or indirectly. To verify that information flows comply with the nontransitive policy, we need to track the dependencies between program variables. In the example, we can easily observe that variable `Bob.data1` depends on `Alice.data` as the information in `Alice.data` may flow to `Bob.data1`. On the other hand, `Bob.data2` does not depend on `Alice.data`. In this paper, we track such dependencies through type inference on information propagation history that is abstracted by security levels. For example, the type system infers that `Bob.data1` depends on the information associated with the security level $\mathbb{S}_\texttt{A}$. The following table summarizes the dependencies of program variables on security levels (including own levels):

$$\texttt{Alice.data} \text{ depends on } \mathbb{S}_\texttt{A}$$
$$\texttt{Bob.data1} \text{ depends on } \mathbb{S}_\texttt{A}, \mathbb{S}_\texttt{B}$$
$$\texttt{Bob.data2} \text{ depends on } \mathbb{S}_\texttt{B}$$

Our security type system then ensures that there must be a flow relation ($\trianglerighteq$) defined in the policy from the *inferred* dependencies of the sender variable to the *specified* label of the receiver variable. For example, in `Bob.bad`, information flows from `Bob.data1`, which depends on both $\mathbb{S}_\texttt{A}$ and $\mathbb{S}_\texttt{B}$, to `Charlie.data`, which is given the label $\mathbb{S}_\texttt{C}$. The type system rejects `Bob.bad` because there is no derivable flow relation $\mathbb{S}_\texttt{A} \trianglerighteq \mathbb{S}_\texttt{C}$. On the other hand, `Bob.good` is safe, where information flows from `Bob.data2` to `Charlie.data`, because `Bob.data2` only depends on $\mathbb{S}_\texttt{B}$ and the policy provides $\mathbb{S}_\texttt{B} \trianglerighteq \mathbb{S}_\texttt{C}$.

Technically nontransitive security policies provide a richer set of information flow relations (than transitive policies) to allow users to easily specify minimal flow relations needed by the components to perform their tasks. This facilitates the *principle of least privileges* [11], because each component must access only the information and resources that are necessary for its legitimate purpose. For example, nontransitive policies can easily express mutual ($\mathbb{S}_\texttt{A} \trianglerighteq \mathbb{S}_\texttt{B}$, $\mathbb{S}_\texttt{B} \trianglerighteq \mathbb{S}_\texttt{A}$) and cyclic ($\mathbb{S}_\texttt{A} \trianglerighteq \mathbb{S}_\texttt{B}$, $\mathbb{S}_\texttt{B} \trianglerighteq \mathbb{S}_\texttt{C}$, $\mathbb{S}_\texttt{C} \trianglerighteq \mathbb{S}_\texttt{A}$) information flow requirements across security levels, while transitive policies would have to merge all security levels with circular dependencies therefore undesirably over-approximate their security requirements.

In general, nontransitive security policies allow users to express meaningful coarse-grained information flow requirements that are otherwise difficult with transitive policies. On the one hand, users do not have to trust the developers of downloaded code components or developer-provided security specification; instead, users can specify own security policies coarsely according to the expected functionalities of the components and enforce these policies through a security type

system. On the other hand, developers do not have to specify fine-grained security policies on their components, reducing considerable annotation overhead. Furthermore, nontransitive policies are more adaptable to policy changes (when security policies are extended, composed or revoked), because all permitted information flow relations are explicitly specified and do not depend on other relations. In contrast, changes made on transitive policies may accidentally introduce undesirable flow relations or remove desirable flow relations that are implicitly inferred thus dependent on added or removed relations.

To summarize, the technical contributions of this paper are:

- We propose a new IFC system that provides guarantees about information propagation in computer software using nontransitive security policies. Informally, our security system ensures that, for any operation, all the code responsible for (i.e. may directly or transitively influence) the operation is explicitly authorized by the given security policy. Formally, this system yields a new noninterference theorem that offers more flexible information flow relations induced by security policies that do not have to be transitive, generalizing the conventional transitive noninterference theorem without distinguishing confidentiality and integrity.
- We present a security type system to demonstrate how to statically enforce security policies that are possibly nontransitive, controlling both explicit and implicit information flows raised from conditionals and lambda values in a small core language with references. Unlike traditional security types that use subtyping (of security levels labeled on program variables) to regulate transitive information propagation across the program, our type system tracks the dependencies of program variables by inferring information flow history in the program and ensuring they respect the specified nontransitive policy. We prove both the subject reduction theorem and the generalized noninterference theorem.

The rest of the paper is organized as follows. Section II describes our core language and its operational semantics, which allow us to formalize the generalized noninterference in Section III. Section IV presents the security type system that statically enforces nontransitive security policies. The noninterference result enforced by the type system is proved in Section V. Section VI discusses an application of nontransitive IFC to the well-known confused deputy problem. Section VII discusses related work and Section VIII concludes the paper.

II. A Core Language

The syntax of our language, defined in Fig. 2, is similar to [9], [12]. A value may be a number, a lambda (standard: a local variable and an expression), or a tagged runtime address ι_α (also called concrete location). We use α to denote the allocation site (or abstract location) of the new memory allocated at ι (we sometimes omit α in rules where it is not used). An expression may be values, variables, function applications, memory allocations (tagged with allocation site α), assignments, dereferences, conditionals or binary operations. This language is intentionally minimal, in order to clearly present our information flow evaluation strategies. A rich variety of additional constructs such as let, sequence or recursion are not directly defined because they are easily encoded [4]. For example, some standard encoding:

$$\begin{array}{rcl} \texttt{let}\ x = e\ \texttt{in}\ e' & \stackrel{\text{def}}{=} & (\lambda x.e)\ e' \\ e; e' & \stackrel{\text{def}}{=} & (\lambda x.e')\ e \text{ where } x \notin FV(e') \end{array}$$

Security policies use the syntax defined in Fig. 3.

$$\begin{array}{llll} \ell & \in & \textit{Label} & \\ \mathcal{P} & \in & \textit{Security Policy} & = \quad \textit{Alloc. Site} \to \textit{Label} \\ \mathcal{S} & \in & \textit{Security Env.} & ::= \quad \emptyset \mid \mathcal{S}, \ell \trianglerighteq \ell \end{array}$$

Fig. 3. Policy syntax.

As usual, we use labels to denote security levels. Each allocation site is assigned to a security label by $\mathcal{P}$, which maps allocation sites to security labels. For example: $\mathcal{P}(\alpha)=\ell$ means the memory allocated at α is assigned to the security label ℓ. A security policy is defined as a pair of $\mathcal{S}$ and $\mathcal{P}$. The security environment $\mathcal{S}$ represents a permitted flow relation $\trianglerighteq \subseteq \textit{Label} \times \textit{Label}$, which is not necessarily transitive. We use the judgment $\mathcal{S} \vdash \ell \trianglerighteq \ell'$ (or simply $\ell \trianglerighteq \ell'$, since there is only one fixed $\mathcal{S}$ in the program) to denote two security labels ℓ and ℓ' are related by $\mathcal{S}$, so that information may flow from memories labeled by ℓ to memories labeled by ℓ'.

We do not explicitly define coarse-grained constructs for our language, because our type system supports both fine-grained and coarse-grained security policies. For coarse-grained policies, $\mathcal{P}$ simply assigns a single security label to all allocation sites found in a code component or module. For example, $\mathcal{P}(\alpha) = \mathcal{P}(\beta)$ when α and β are allocation sites in the same component.

The evaluation rules are defined in the form as follows:

$$M\ e \Downarrow M'\ v$$

$$\frac{}{M\ v \Downarrow M\ v} \quad \text{[VAL]}$$

$$\frac{M\ e \Downarrow M'\ \lambda x.e'' \qquad M'\ e' \Downarrow M''\ v \qquad M''\ e''[v/x] \Downarrow M'''\ v'}{M\ e\ e' \Downarrow M'''\ v'} \quad \text{[APP]}$$

$$\frac{M\ e \Downarrow M'\ c \qquad c \neq 0 \qquad M'\ e' \Downarrow M''\ v}{M\ (\texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'') \Downarrow M''\ v} \quad \text{[THEN]}$$

$$\frac{M\ e \Downarrow M'\ 0 \qquad M'\ e'' \Downarrow M''\ v}{M\ (\texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'') \Downarrow M''\ v} \quad \text{[ELSE]}$$

$$\frac{M\ e \Downarrow M'\ v \qquad \iota_\alpha \notin dom(M')}{M\ \texttt{new}_\alpha\ e \Downarrow (M', \iota_\alpha \mapsto v)\ \iota_\alpha} \quad \text{[ALLOC]}$$

$$\frac{M\ e \Downarrow M'\ \iota \qquad M'\ e' \Downarrow M''\ v}{M\ e := e' \Downarrow M''[\iota \mapsto v]\ v} \quad \text{[ASSIGN]}$$

$$\frac{M\ e \Downarrow M'\ \iota \qquad M'(\iota) = v}{M\ !e \Downarrow M'\ v} \quad \text{[DEREF]}$$

$$\frac{M\ e \Downarrow M'\ c \qquad M'\ e' \Downarrow M''\ c'}{M\ e \oplus e' \Downarrow M''\ c \oplus c'} \quad \text{[OP]}$$

Fig. 4. Evaluation rules.

where an expression e is evaluated in the context of a memory M, and it returns the resulting value v and (possibly modified) M'. The memory (or store), M, maps addresses to values:

$$M \quad \in \quad \textit{Memory} \quad = \quad \textit{Address} \rightarrow \textit{Value}$$

A big-step operational semantics is given in Fig. 4, which is mostly standard. In [APP], instead of keeping bindings, function parameter x is substituted upon application (the notation $e[v/x]$ denotes x is substituted for v in the term e). [ASSIGN] assigns the value that e' evaluates to, to the location that e evaluates to (the notation $M[\iota \mapsto v]$ denotes ι is updated in M with the value v). [ALLOC] extends the memory with a newly allocated address. Note that the choice of fresh address allocated to the new memory allocation is arbitrary in [ALLOC]. This allocation behavior may be accommodated using standard renaming of locations over an indistinguishability relation on memories (in Section III). But for simplicity, like in [2], [13], we assume a deterministic scheme for memory allocation. We sometimes write $\mathcal{P}(\iota_\alpha){=}\ell$ if $\mathcal{P}(\alpha){=}\ell$ for some $\ell \in \textit{Label}$. An address ι is associated with a security label ℓ where $\mathcal{P}(\iota) = \ell$, and may serve as a communication channel for the security level ℓ. In that sense, a value (input) can be left at ι before the program execution and its resulting value (output) can be observed at the end of the execution.

III. Nontransitive Noninterference

In this section we define the main security property of our IFC system, nontransitive noninterference, provided by security policies that are not necessarily transitive. Our notion of nontransitive noninterference is not to be confused with the previous notion of intransitive noninterference [14]–[19], which has a completely different interpretation. Intransitive noninterference is a flow-control mechanism which controls the *path* of information flow with respect to the various security levels of the system. The canonical example is a confidentiality policy which says that information may flow directly from low-level A to high-level C (the transitive part of the policy), and from C to a declassifier level B, and from B to A, but not directly from C to A (the intransitive part of the policy). The definition of intransitive noninterference ensures that any downgraded information indeed passes through the declassifier, and is thereby controlled; it is classified as a type of where-based downgrading methods by [20]. On the other hand, our nontransitive noninterference enforces that any (direct or indirect) information flow from two security levels must be explicitly specified by the security policies. For example, if a flow relation from A to C is not specified in the policy, then no information can flow from A to C, neither directly (from A to C) nor through other security levels (e.g. from A to B and from B to C). Thus, intransitive noninterference is an extension of the standard transitive noninterference (i.e. lattice-based flow relations) with intransitive downgrading exceptions, while nontransitive noninterference is a generalization of transitive noninterference where all flow relations (including transitive closures of transitive relations) are explicitly specified in the policy.

With the formal notion of noninterference by Goguen and Meseguer [1], a system is modeled as a state machine with inputs and outputs classified by security levels. The noninterference property for a confidentiality policy guarantees that any sequence of lower-level inputs will produce the same lower-level outputs, regardless of what the higher-level inputs are. Intuitively, this ensures that an attacker at lower-level is unable to distinguish two computations from their outputs if they vary only in their higher-level secret inputs. Conversely, the noninterference property for an integrity policy guarantees any sequence of higher-level inputs will produce the same higher-level outputs, regardless of what the lower-level inputs are. This ensures that a system keeps its trusted contents/outputs (at higher-level) unaltered by any untrusted (lower-level) inputs.

In our system, given $\ell \in \textit{Label}$, for any security label ℓ', we have either $\ell' \trianglerighteq \ell$ or $\ell' \not\trianglerighteq \ell$. Although there is no ordering (*high* and *low*) of security levels in nontransitive policies, the Goguen and Meseguer-style noninterference [1] may be considered still valid in the sense that the outputs observable by ℓ may only depend on the inputs with security labels permitted to interfere with ℓ, therefore altering inputs from a label ℓ' that is not permitted to interfere with ℓ does not cause any change to the outputs of ℓ. In a language-based setting, memory addresses assigned to different security labels become channels for inputs and outputs, and the notion of noninterference can be captured by an indistinguishability relation on memories with respect to specific security labels.

The formalism of our noninterference property uses a

number of auxiliary definitions. Two memory states are indistinguishable for a specified security label ℓ if all memory addresses with the label ℓ contain the same content in both states. We define an indistinguishability relation for both a single security label $\ell \in \mathit{Label}$ and a set of security labels $\mathcal{L} \subseteq \mathit{Label}$ respectively, as follows.

$$\frac{\forall \iota \in dom(M_1) \cup dom(M_2) \cdot \mathcal{P}(\iota) = \ell \Longrightarrow M_1(\iota) = M_2(\iota)}{M_1 \overset{\ell}{=} M_2} [\ell\text{-EQ}]$$

$$\frac{\forall \ell \in \mathcal{L} \cdot M_1 \overset{\ell}{=} M_2}{M_1 \overset{\mathcal{L}}{=} M_2} [\mathcal{L}\text{-EQ}]$$

Given a deterministic memory allocation scheme, two different runs of a program will assign the same address value if the same program counter $\texttt{new}_\alpha$ is evaluated the same number of times. As a consequence, we have the following property.

Lemma 1:

$$\left.\begin{array}{c} M_1 \overset{\mathcal{P}(\alpha)}{=} M_2 \\ M_1\ \texttt{new}_\alpha\ v_1 \Downarrow M_1'\ \iota_1 \\ M_2\ \texttt{new}_\alpha\ v_2 \Downarrow M_2'\ \iota_2 \end{array}\right\} \Longrightarrow \iota_1 = \iota_2$$

If M_1 and M_2 agree on the security label $\mathcal{P}(\alpha)$ that is assigned to memory allocation α, then for all $\iota_\alpha \in dom(M_1)$, we have $\iota_\alpha \in dom(M_2)$, and for all $\iota_\alpha \in dom(M_2)$, we have $\iota_\alpha \in dom(M_1)$. Therefore, M_1 and M_2 contain the same number of references to allocations at α. Then by the deterministic allocation scheme, ι_1 freshly allocated at α in M_1 must be the same as ι_2 freshly allocated at α in M_2.

Given $\ell \in \mathit{Label}$ and an information flow relation $\trianglerighteq$ that is not necessarily transitive, we collect all security labels that are permitted to pass information to ℓ in a set $\trianglerighteq\ell = \{\ell' \mid \ell' \trianglerighteq \ell\}$. No security label outside $\trianglerighteq\ell$ is allowed to interfere with ℓ. We also have $\ell \in \trianglerighteq\ell$ because the flow relation is reflexive. Therefore, the security requirement for the label ℓ is that the final outputs observable by ℓ only depend on inputs from security labels that are permitted to (both directly and indirectly) interfere with ℓ (i.e., the set $\trianglerighteq\ell$). Now, we can present our termination-insensitive noninterference property using memory indistinguishability.

Definition 1: (Nontransitive noninterference)

$$\left.\begin{array}{c} M_1 \overset{\trianglerighteq\ell}{=} M_2 \\ M_1\ e \Downarrow M_1'\ v \\ M_2\ e \Downarrow M_2'\ v \end{array}\right\} \Longrightarrow M_1' \overset{\ell}{=} M_2'$$

The notion of nontransitive noninterference formalized in *Definition 1* states that any sequence of inputs will produce the same outputs, as long as all inputs that may influence them are initially the same (i.e. regardless of any other inputs that may not influence them, which may differ freely). Nontransitive noninterference implies that information flows, either directly or indirectly, are possible only if their flow relations are specified explicitly by the nontransitive policy $\trianglerighteq$. This property makes it easy to control the extent of information flows in the program.

Transitive noninterference provides either an integrity property or a confidentiality property due to the ordering of security levels. The attacker and the system are typically considered lower-level and higher-level respectively, so that information is permitted to flow in only one direction: from lower-level to higher-level in confidentiality noninterference, alternatively, from higher-level to lower-level in integrity noninterference. On the other hand, without security level ordering, nontransitive noninterference does not distinguish confidentiality and integrity therefore uniformly provides both. The explicit flow relation provided by $\trianglerighteq$ governs both ends of any information flow, which can be either the attacker or the system. Intuitively, our property ensures that an attacker keeps its untrusted contents/outputs unaltered by any secret inputs, at the same time, a system keeps its secret contents/outputs unaltered by any untrusted inputs.

IV. The Type System

In this section, we show how to enforce nontransitive noninterference by using a security type system. Like transitive security type systems, we use security labels to abstract program values for tracking dependencies between them. Unlike transitive security type systems that verify transitive policies by subtyping security labels associated with program variables, we track dependencies by capturing the history of information flows through security labels defined by nontransitive policies.

τ	$\in$	*Labeled Type*	$::=$	$t^{\mathcal{L}}$
t	$\in$	*Unlabeled Type*	$::=$	$\texttt{n} \mid \tau \xrightarrow{\mathcal{L}} \tau \mid \texttt{ref}_{\mathcal{A}}\ \tau$
$\mathcal{A}, \mathcal{B}$	$\in$	*Set of Alloc. Sites*	$::=$	$\emptyset \mid \mathcal{A}, \alpha$
$\mathcal{L}, pc$	$\in$	*Set of Labels*	$::=$	$\emptyset \mid \mathcal{L}, \ell$

Fig. 5. Type syntax.

Types are defined in Fig. 5, where every type τ, including a type nested inside another, carries a set of security labels. In the absence of label transitivity, a set of labels $\mathcal{L}$ (rather than a single label) is used to represent the dependencies of the values the types ascribe. Security labels (denoted by ℓ) are drawn from an arbitrary relation $\trianglerighteq$, which is not necessarily transitive. In our typing rules, we use $\mathcal{L}_1 \trianglerighteq \mathcal{L}_2$ to denote that $\ell_1 \trianglerighteq \ell_2$ for all $\ell_1 \in \mathcal{L}_1$ and $\ell_2 \in \mathcal{L}_2$, where $\mathcal{L}_1, \mathcal{L}_2 \subseteq \mathit{Label}$.

For convenience, we also define unlabeled types for base, function and reference. We use $\texttt{n}$ to denote the base type for constant values. The write effect $\mathcal{L}$ of a lambda type $\tau \xrightarrow{\mathcal{L}} \tau$ is captured by a label set that tracks all possible implicit flows to the body of the function from control flow structures. The reference type $\texttt{ref}_{\mathcal{A}}$ carries a set of allocation sites $\mathcal{A}$, representing all possible allocation sites of the memory. Note that all values produced within the scope of the same component implicitly have that label; hence, it is not necessary to label individual values (as explained previously, $\mathcal{P}(\alpha)$ is used to look up the specified label for the value created at allocation site α).

The subtyping rules are defined in Fig. 6. The label set in labeled types can be weakened (increased) freely with

the subtyping rule [S-TYPE]. Subtyping for unlabeled reference types $\texttt{ref}_{\mathcal{A}}\ \tau$ is invariant in τ, as usual, but co-variant in the set of possible allocation sites $\mathcal{A}$ (i.e. over-approximation). Subtyping for unlabeled function types is co-variant in output type and contra-variant in input type and function effect.

$$\frac{t \leq t' \qquad \mathcal{L} \subseteq \mathcal{L}'}{t^{\mathcal{L}} \leq t'^{\mathcal{L}'}} \quad \text{[S-TYPE]}$$

$$\frac{\mathcal{A} \subseteq \mathcal{B}}{\texttt{ref}_{\mathcal{A}}\ \tau \leq \texttt{ref}_{\mathcal{B}}\ \tau} \quad \text{[S-REF]}$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad \mathcal{L}' \subseteq \mathcal{L}}{\tau_1 \xrightarrow{\mathcal{L}} \tau_2 \leq \tau_1' \xrightarrow{\mathcal{L}'} \tau_2'} \quad \text{[S-FUN]}$$

Fig. 6. Subtyping rules.

The type judgment is defined as follows:

$$\Gamma\ \Pi \vdash_{pc} e : \tau$$

The type environment Γ maps free variables of e to their types. Another type environment Π provides allocation typing, mapping allocation sites to the types of contents in the allocated memory. To help formalize and prove the safety properties, Π predicts the types for all allocation sites and must be respected by all actual values at runtime (see [MEMORY] in Section V). Our type judgment means that, given the types for free variables in Γ and allocations in Π, e has type τ.

The annotation pc, often called the program counter label, tracks implicit information flows from program structures to the write effects of e. The type system ensures that for any reference that e writes, there must be a flow relation by the security policy from pc to the specified label of the reference's allocation site (see [T-ASN] and [T-NEW]). This is necessary to prevent information leaks via the heap.

The typing rules are shown in Fig. 7, which track dependencies of values in types ($t^{\mathcal{L}}$) by collecting all security labels of historical flows into the label set $\mathcal{L}$ carried by their types. We explain the important rules in more detail. The standard subsumption rule is defined in [T-SUB], using the subtyping rules. For simplicity, we use $\tau^{\cup\mathcal{L}}$ to denote $t^{\mathcal{L}'\cup\mathcal{L}}$ where $\tau = t^{\mathcal{L}'}$.

Dereferences read information from memory. The rule for dereference [T-DEREF] looks up the security types of all possible allocation sites of the reference and ensures they are subsumed by the resulting type of the dereference. This implies that all labels collected from historical information flows into the reference are captured in the label set of the resulting type (because subtyping preserves the set of labels). Note that the reference type $\texttt{ref}_{\mathcal{A}}$ carries a set of allocation sites, representing all possible allocation sites of the memory that e is evaluated to.

Assignments write information to memory. The rule for assignment [T-ASN], similarly, tracks information flows from e' to all possible allocation sites of the reference type. In addition, it checks that such information flows are permitted for all possible allocation sites by flow relations defined by $\trianglerighteq$.

$$\frac{\Gamma\ \Pi \vdash_{pc'} e : \tau' \qquad pc \subseteq pc' \qquad \tau' \leq \tau}{\Gamma\ \Pi \vdash_{pc} e : \tau} \quad \text{[T-SUB]}$$

$$\frac{}{\Gamma\ \Pi \vdash_{pc} c : \mathtt{n}^{\mathcal{L}}} \quad \text{[T-CONST]}$$

$$\frac{\Gamma, x : \tau'\ \Pi \vdash_{\mathcal{L}'} e : \tau \qquad \mathcal{L} \subseteq \mathcal{L}'}{\Gamma\ \Pi \vdash_{pc} \lambda x.e : (\tau' \xrightarrow{\mathcal{L}'} \tau)^{\mathcal{L}}} \quad \text{[T-FUN]}$$

$$\frac{\Gamma(x) = \tau}{\Gamma\ \Pi \vdash_{pc} x : \tau} \quad \text{[T-VAR]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : (\tau' \xrightarrow{\mathcal{L}'} \tau)^{\mathcal{L}} \qquad \Gamma\ \Pi \vdash_{pc} e' : \tau' \qquad pc \subseteq \mathcal{L}'}{\Gamma\ \Pi \vdash_{pc} e\ e' : \tau^{\cup\mathcal{L}}} \quad \text{[T-APP]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : \mathtt{n}^{\mathcal{L}} \qquad \Gamma\ \Pi \vdash_{pc\cup\mathcal{L}} e' : \tau \qquad \Gamma\ \Pi \vdash_{pc\cup\mathcal{L}} e'' : \tau}{\Gamma\ \Pi \vdash_{pc} \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' : \tau^{\cup\mathcal{L}}} \quad \text{[T-IF]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : t^{\mathcal{L}} \qquad t^{\mathcal{L}\cup pc} \leq \Pi(\alpha) \qquad \mathcal{L} \cup pc \trianglerighteq \mathcal{P}(\alpha) \qquad \mathcal{P}(\alpha) \in \mathcal{L}'}{\Gamma\ \Pi \vdash_{pc} \texttt{new}_{\alpha}\ e : (\texttt{ref}_{\{\alpha\}}\ t^{\mathcal{L}})^{\mathcal{L}'}} \quad \text{[T-NEW]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : (\texttt{ref}_{\mathcal{A}}\ t^{\mathcal{L}})^{\mathcal{L}'} \qquad \Gamma\ \Pi \vdash_{pc} e' : t^{\mathcal{L}'} \qquad \forall \alpha \in \mathcal{A}\ \cdot\ \mathcal{L}' \cup pc \trianglerighteq \mathcal{P}(\alpha) \quad t^{\mathcal{L}\cup\mathcal{L}'\cup pc} \leq \Pi(\alpha)}{\Gamma\ \Pi \vdash_{pc} e := e' : t^{\mathcal{L}'}} \quad \text{[T-ASN]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : (\texttt{ref}_{\mathcal{A}}\ \tau)^{\mathcal{L}} \qquad \forall \alpha \in \mathcal{A}\ \cdot\ \Pi(\alpha) \leq \tau^{\cup\mathcal{L}}}{\Gamma\ \Pi \vdash_{pc} !e : \tau^{\cup\mathcal{L}}} \quad \text{[T-DEREF]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} e : \mathtt{n}^{\mathcal{L}} \qquad \Gamma\ \Pi \vdash_{pc} e' : \mathtt{n}^{\mathcal{L}}}{\Gamma\ \Pi \vdash_{pc} e \oplus e' : \mathtt{n}^{\mathcal{L}}} \quad \text{[T-OP]}$$

$$\frac{\Gamma\ \Pi \vdash_{pc} \Pi(\alpha) = \tau}{\Gamma\ \Pi \vdash_{pc} \iota_{\alpha} : \texttt{ref}_{\{\alpha\}}\ \tau} \quad \text{[T-ADDR]}$$

Fig. 7. Typing rules.

Significantly, nontransitive IFC is enabled by separating the permissible reads (determined by the labels *inferred* from historical flows, i.e. $\mathcal{L}$ where $t^{\mathcal{L}} = \Pi(\alpha)$) and writes (determined by the label *specified* by pre-defined security policy, i.e. $\mathcal{P}(\alpha)$) of the references. Both reads and writes are governed by a nontransitive relation $\trianglerighteq$, allowing us to escape from the strong transitivity of traditional security types.

The rule for memory allocation [T-NEW] tracks information flow from e to the allocation site α in Π and ensures the flow is permitted by flow relations defined by $\trianglerighteq$. The specified security label of the abstract location α is captured in the

label set of the resulting type.

In the rule for constant [T-CONST], we allow a constant value to be given any security label (unbound in the rules) because, intuitively, a value is never intrinsically sensitive—it is sensitive only if it comes from a sensitive location [3]. In the rule for function [T-FUN], $\mathcal{L}$ is covered in $\mathcal{L}'$ to prevent implicit leaks arising from the identity of the function when it gets evaluated. The rule for application [T-APP] ensures that pc is carried into the write effect of the function $\mathcal{L}'$.

In the rule for conditional [T-IF], both branches are typed in a pc that is joined with $\mathcal{L}$, which is the label set of the condition e. This ensures that the execution of the branches is control dependent on a value produced by the condition.

The rule for runtime address [T-ADDR] ensures that the address has the type predicted by the allocation typing Π. Of course, these typing rules will accurately predict the results of evaluation only if the concrete memory used during evaluation actually conforms to the allocation typing that we assume for purposes of type checking (see [MEMORY] in Section V).

V. Nontransitive Noninterference by Type Checking

In this section, we prove that a well-typed program satisfies nontransitive noninterference. The following rule defines well-formed memory. Intuitively, a memory M is consistent with an allocation typing Π if every value in the memory has the type predicted by the allocation typing.

$$\frac{\forall \iota_\alpha \ \cdot \ M(\iota_\alpha) = v \implies \Gamma\ \Pi \vdash_{pc} v : \Pi(\alpha)}{\Gamma\ \Pi \vdash_{pc} M} \quad \text{[MEMORY]}$$

Our type system guarantees information that flows into a location labelled by ℓ must be permitted to influence ℓ by the given policies for any program evaluation. This is represented by the indistinguishability relation on memory states related by $\overset{\ell}{=}$, provided that all information allowed to interfere with ℓ (i.e., $\trianglerighteq \ell$) is the same before the evaluation and both evaluations terminate. The main result of the paper is stated in *Theorem 1*, which enforces *Definition 1* by typing.

Theorem 1: (Nontransitive noninterference by type system)

$$\left.\begin{array}{c} M_1 \overset{\trianglerighteq \ell}{=} M_2 \\ M_1\ e \Downarrow M_1'\ v \\ M_2\ e \Downarrow M_2'\ v \\ \Gamma\ \Pi \vdash_{pc} e : t^{\{\ell\}} \\ \Gamma\ \Pi \vdash_{pc} M_1 \\ \Gamma\ \Pi \vdash_{pc} M_2 \end{array}\right\} \implies M_1' \overset{\ell}{=} M_2'$$

In order to prove *Theorem 1*, we need to establish a few intermediate lemmas. As usual, we show that if an expression is well typed, then its type is preserved after evaluation.

Lemma 2: (Subject reduction)

$$\left.\begin{array}{c} \Gamma\ \Pi \vdash_{pc} e : t^{\mathcal{L}} \\ \Gamma\ \Pi \vdash_{pc} M \\ M\ e \Downarrow M'\ v \end{array}\right\} \implies \left\{\begin{array}{l} \Gamma\ \Pi \vdash_{pc} v : t^{\mathcal{L}} \\ \Gamma\ \Pi \vdash_{pc} M' \end{array}\right.$$

Proof: To show the preservation of type from e to v, we prove by structural induction on e. We only show the case of [T-APP], as the other cases are more straightforward.

Let $M\ e_1\ e_2 \Downarrow M'''\ v$ then by definition we have $M\ e_1 \Downarrow M'\ \lambda x.e$, $M'\ e_2 \Downarrow M''\ v'$ and $M''\ e[v'/x] \Downarrow M'''\ v$. Let $\Gamma\ \Pi \vdash_{pc} e_1\ e_2 : t^{\mathcal{L}}$.

Then by [T-APP] we have

$\Gamma\ \Pi \vdash_{pc} e_1 : (\tau' \xrightarrow{\mathcal{L}'} t^{\mathcal{L}_1})^{\mathcal{L}_2}$, (1)
$\Gamma\ \Pi \vdash_{pc} e_2 : \tau'$, (2)
$pc \subseteq \mathcal{L}'$, (3)
$\mathcal{L}_1 \cup \mathcal{L}_2 = \mathcal{L}$. (4)
By I.H. and (1), we have
$\Gamma\ \Pi \vdash_{pc} \lambda x.e : (\tau' \xrightarrow{\mathcal{L}'} t^{\mathcal{L}_1})^{\mathcal{L}_2}$. (5)
Similarly, by I.H. and (2),
$\Gamma\ \Pi \vdash_{pc} v' : \tau'$. (6)
Then by [T-FUN] and (5), we have
$\Gamma, x : \tau'\ \Pi \vdash_{L'} e : t^{\mathcal{L}}$. (7)
Therefore $\Gamma\ \Pi \vdash_{\mathcal{L}'} e[v'/x] : t^{\mathcal{L}}$. (8)
Then by I.H. and (8), we have
$M''\ e[v'/x] \Downarrow M'''\ v$ and $\Gamma\ \Pi \vdash_{\mathcal{L}'} v : t^{\mathcal{L}}$. (9)
By (3) and [T-SUB], $\Gamma\ \Pi \vdash_{pc} v : t^{\mathcal{L}}$. (10)

The preservation of well-formed memory is by [T-NEW] and [T-ASN]. □

The next lemma says if an expression evaluates into a memory location, then the information about its allocation site is stored in its type.

Lemma 3: (Location)

$$\left.\begin{array}{c} \Gamma, \Pi \vdash_{pc} e : (\mathtt{ref}_{\mathcal{A}}\tau)^{\mathcal{L}} \\ M\ e \Downarrow M'\ \iota_\alpha \end{array}\right\} \implies \alpha \in \mathcal{A}$$

Proof: By *Lemma* 2 (Subject reduction), we have $\Gamma, \Pi \vdash_{pc} \iota_\alpha : (\mathtt{ref}_{\mathcal{A}}\tau)^{\mathcal{L}}$, which implies that $\mathtt{ref}_{\{\alpha\}}\tau$ is a subtype of $\mathtt{ref}_{\mathcal{A}}\tau$, i.e., $\{\alpha\} \subseteq \mathcal{A}$ by [S-REF]. □

Given $\Pi(\alpha)$ a type carrying the set of security labels that may flow to allocation α, we present the following intermediate result regarding the property of function Π. We write $\widehat{\Pi}(\alpha)$ for the set of labels $\mathcal{L}$ if $\Pi(\alpha) = t^{\mathcal{L}}$.

Two security labels ℓ_1 and ℓ_2 are effectively equivalent if (1) $\ell_1 \trianglerighteq \ell_2$, (2) $\ell_2 \trianglerighteq \ell_1$, (3) $\ell \trianglerighteq \ell_1$ iff $\ell \trianglerighteq \ell_2$ for all ℓ, (4) $\ell_1 \trianglerighteq \ell$ iff $\ell_2 \trianglerighteq \ell_1$ for all ℓ. This may be used to encode allocation sites within the sample code block by equivalent security labels. Since our system allows security labels to be effectively equivalent yet still different, without loss of generality, to this point, we assume that all allocations have distinct labels in the proofs.

Lemma 4: (Transitivity with respect to Π)

$$\ell \in \widehat{\Pi}(\alpha) \wedge \mathcal{P}(\alpha) \in \widehat{\Pi}(\beta) \implies \ell \in \widehat{\Pi}(\beta)$$

Proof: Suppose $\ell \in \widehat{\Pi}(\alpha)$ and $\mathcal{P}(\alpha) \in \widehat{\Pi}(\beta)$. The only ways to have a security label included in $\widehat{\Pi}(\beta)$ are via [T-NEW] or [T-ASN]. Without loss of generality, we assume it is the case of [T-ASN]. That is, we have $\Gamma\ \Pi \vdash_{pc} e := e' : t^{\mathcal{L}'}$, $\Gamma\ \Pi \vdash_{pc} e : (\mathtt{ref}_{\mathcal{A}}\ t^{\mathcal{L}})^{\mathcal{L}'}$, $\Gamma\ \Pi \vdash_{pc} e' : t^{\mathcal{L}'}$, e evaluates to ι_β with $\beta \in \mathcal{A}$, and $\mathcal{L}' \cup pc \subseteq \widehat{\Pi}(\beta)$. Therefore, either $\mathcal{P}(\alpha) \in \mathcal{L}'$ or $\mathcal{P}(\alpha) \in pc$.

Without loss of generality, assume $\mathcal{P}(\alpha) \in \mathcal{L}'$. Then the only way to have $\mathcal{P}(\alpha)$ included in $\mathcal{L}'$ is via [T-DEREF], in the way that $\widehat{\Pi}(\alpha) \subseteq \mathcal{L}'$, which derives $\widehat{\Pi}(\alpha) \subseteq \widehat{\Pi}(\beta)$. Then $\ell \in \widehat{\Pi}(\beta)$. □

Intuitively, by the rules [T-DEREF], [T-ASN] and [T-NEW], if there is a flow from location α to location β, all security labels stored in $\widehat{\Pi}(\alpha)$ will be copied to $\widehat{\Pi}(\beta)$ either directly or indirectly.

Moreover, given a security label ℓ, only those values labeled by ℓ' satisfying $\ell' \trianglerighteq \ell$ are allowed to influence an address labeled by ℓ in a typable program, as witnessed in [T-NEW] and [T-ASN]. Define $\widehat{\Pi}(\ell) = \bigcup_{\mathcal{P}(\alpha)=\ell} \widehat{\Pi}(\alpha)$, then we have the following property.

Lemma 5: For all $\ell \in$ *Label*, $\widehat{\Pi}(\ell) \trianglerighteq \ell$.

Proof: Similar to the above lemma, this is a direct consequence of [T-NEW] and [T-ASN]. □

For the next lemma, if expression e is typable at program counter pc, then the evaluation of e does not modify memory locations that are not influenced by (labels in) pc. Recall that in a transitive policy (such as in [9]), the type system enforces a lower bound type pc as the permitted write effects of e. For example, given the two-level policy $l \trianglerighteq h$, $\Gamma, \Pi \vdash_{\{h\}} e$ restricts all write effects to the h-component of a memory state, and the evaluation of $M\ e \Downarrow M'\ v$ results in a pair of l-equivalent memory states M and M'. We extend this way of thinking to the more general (nontransitive) policies, such that the type system enforces an equivalence relation for all labels not (directly) influenced by security labels in pc. For the sake of readability, we present the proof of *Lemma* 6 in the appendix.

Lemma 6: (Local-Respects)

$$\left.\begin{array}{c} \Gamma, \Pi \vdash_{pc} e : t^{\mathcal{L}} \\ pc \not\subseteq \widehat{\Pi}(\ell) \\ M\ e \Downarrow M'\ v \end{array}\right\} \Longrightarrow M \overset{\widehat{\Pi}(\ell)}{=} M'$$

The next lemma serves as the main stepping stone to the proof of Theorem 1, which says that given a typable expression which allows inputs (by substitution), the final result of its evaluation is independent to inputs that are not entirely labeled in $\widehat{\Pi}(\ell)$, and the indistinguishability relation on memories with respect to $\widehat{\Pi}(\ell)$ is maintained during evaluation. Again, since the proof of *Lemma* 7 is quite involved, which applies the results of most previous lemmas, we leave the complete proof in the appendix for better readability of the paper.

Lemma 7: (Substitution consistency)

$$\left.\begin{array}{c} M_1 \overset{\widehat{\Pi}(\ell)}{=} M_2 \\ M_1\ e[\vec{u}/\vec{x}] \Downarrow M_1'\ v_1' \\ M_2\ e[\vec{v}/\vec{x}] \Downarrow M_2'\ v_2' \\ \Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e : t_2^{\mathcal{L}} \\ \forall j \in I.\ \mathcal{L}_j \subseteq \widehat{\Pi}(\ell) \Rightarrow u_j = v_j \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2' \\ \\ \mathcal{L} \subseteq \widehat{\Pi}(\ell) \\ \quad \Rightarrow v_1' = v_2' \end{array}\right.$$

$e[\vec{u}/\vec{x}]$ is short for the substitution $e[u_1/x_1, \ldots u_{|I|}/x_{|I|}]$, where I is a finite index.

Finally, given $\ell \subseteq \widehat{\Pi}(\ell)$ (by reflexivity of a policy) and $\widehat{\Pi}(\ell) \subseteq \trianglerighteq \ell$ (by *Lemma* 5), by treating e as an empty input vector $\vec{\epsilon}$, a special form of the premises for *Lemma* 7, the result of *Theorem* 1 directly follows.

```
Library { // protected code
          // only accessible via Service
  process(source_data){ ... }
  retrieve(key){
    ...
    ret some_data;
  }
}

Service { // priviledged code
  logFile;
  addLog(x) {
    append(logFile, x);
  }
  print(data) {
    Library.process(data);
  }
  query_data(key){
    ret Library.retrieve(key);
  }
}

Downloaded_Code { // untrusted code
  d = ...;        // untrusted data
  main() {
    Service.addLog(d);
    Service.print(d.data);
    ... = Service.query_data(d.key);
  }
}

Trusted_Code { // trusted local code
  ...
}
```

Fig. 8. Confused deputy attack.

VI. Example: Confused Deputy Problem

In addition to avoiding unwanted transitive information flows and supporting a richer set of information flow relations, our nontransitive IFC system has significant implications for the programming model and usability. In particular, it provides meaningful coarse-grained security policies for effectively reasoning about high-level interactions between program components, separated from their implementation details. This means the users of untrusted code do not have to trust or verify developer-provided specification; instead, users can specify high-level security policies on untrusted code and let the type system perform low-level analysis to automatically enforce these policies. The separation of security policies and implementation details of software components strengthens the security and distribution of responsibility, as well as promoting the reusability of components by favoring use-site policies over definition-site policies. Thus, our approach enables reasoning that is difficult in conventional transitive IFC systems.

In this section, we discuss an application of nontransitive IFC to the *confused deputy problem*, using the example in Fig. 8. We show how to use nontransitive type system to enforce confused deputy attack freedom (CDAF). A confused deputy problem occurs when trusted, more privileged code is manipulated by untrusted code into misusing its authority to perform a protected sensitive action. Rajani et al. [21] define CDAF as information flow *integrity*, where a confused deputy attack is considered as a breach of system integrity. Given an Attacker's Interest Set (AIS, i.e., the set of resources in the system to be protected from untrusted parties), a system is CDAF, if for all locations $r \in AIS$, either the attacker's code can (directly) determine the value of r, or r cannot be (indirectly) influenced by the attacker. In other words, if the attacker is not allowed to modify system resource r, then there is no way for the attacker to make changes of r via an intermediate party (the deputy).

In the example, we assume a local system in which the `Library` is protected and accessible via only the privileged code `Service`. Code downloaded from Internet is not trusted, which may be permitted to access `Service` but not `Library`. For instance, `Downloaded_Code` is allowed to invoke `addLog()` that appends some (untrusted) data into a non-executable log file to be inspected manually by the system administrator. Since the appended data is not further propagated to other parts of the system, this action is regarded as *secure*. The untrusted `Downloaded_Code` also invokes `print()` from `Service` that reads and prints out untrusted data using protected `Library` resources via `process()`. This call is *insecure*. Although the function `print()` has the privilege to call `process()` from `Library`, and `Downloaded_code` has the privilege to use the `Service` interface, `Downloaded_Code` should not influence the execution of `Library` functions. Consequently, this exhibits a confused deputy problem—while `Downloaded_code` cannot directly access the printout in `Library`, it can still influence the printout via the (confused) deputy `Service`.

Transitive IFC with fine-grained security policies can solve the above problem by assigning security levels to individual program variables in `Service`, for example, specifying `addLog.x` to be accessible to all code while `print.data` to be accessible to only *trusted* local code. However, since `Service` may contain a large number of variables and functions to be shared among a variety of third party applications, specification of individual program variables may be difficult and error-prone. Nontransitive IFC with coarse-grained security policies provides a simpler and more intuitive solution, by assigning security levels L, S and D to code bases `Library`, `Service` and `Downloaded_Code` respectively, and specifying $D \trianglerighteq S$ and $S \trianglerighteq L$ (thus disallowing $D \trianglerighteq L$). With such policies, D cannot influence L directly or indirectly, therefore D's access to S must not interfere with the functionalities provided by L. Coarse-grained flow policies simply rule out any untrusted code (e.g. `Downloaded_Code`) that may (indirectly) cause changes to `Library`, achieving CDAF. For any trusted local code such as `Trusted_Code`, we simply assign it a security level T and add $T \trianglerighteq S$ and $T \trianglerighteq L$ to the existing policies consistently.

Note that `Downloaded_Code` also invokes `query_data()` which retrieves information from `Library` via `Service`. This is a violation of information flow confidentiality, and can be rejected by adding additional nontransitive security policies $L \trianglerighteq S$ and $S \trianglerighteq D$ (thus disallowing $L \trianglerighteq D$). Therefore, nontransitive IFC can simultaneously enforce integrity (such as CDAF) as well as prevent breach of confidentiality via the confused deputy.

We briefly discuss how the type system of Section IV can be used to enforce the nontransitive security policies used in this example. First, all variables in `Downloaded_Code` are initially typed $\mathsf{n}^{\{D\}}$ where n is the base type and D is the security label specified to the entire downloaded code. Since `Downloaded_Code` invokes functions in `Service`, it propagates the label D to the variables in `Service`; the type system infers $\Gamma(\texttt{Service.print.data}) = \mathsf{n}^{\{D,S\}}$ where S is the security label specified to `Service`. This label D is further propagated through a call to `Library`, thus $\Gamma(\texttt{Library.process.source\_data}) = \mathsf{n}^{\{D,S,L\}}$ where L is the security label specified to `Library`. Since $D \trianglerighteq L$ is not explicitly specified in the security policies, the composition of `Downloaded_Code` with the local system is not typeable, and the program will be consequently rejected by the type system.

VII. Related Work

We focus on the literature closely related to our work on noninterference with a general flow relation and those of similar semantic frameworks in language-based security.

The notion of noninterference based on information flow is introduced in Goguen and Meseguer's seminal work [1], where the transitivity of flow relations has been discussed. Based on a deterministic state machine model, the authors suggest that an information flow relation is not necessarily transitive and the only restriction placed on the flow relation is that it should be reflexive. However, in their presented formulation, only transitive relations have a useful interpretation. Rushby [15] considers an interpretation of intransitive policies that specify channel-control behaviors, such that information can flow upward in security level without restriction (a standard transitive policy), but only flow downward through the mediation of the presumably trusted declassifier domain (a where-based downgrading policy [20]). The notation of intransitive noninterference is developed with unwinding conditions in a state-based system model. A similar notion of noninterference is proposed by Pinsky [16], which also uses deterministic state machines as system model. [18] focuses on definitions that can cope with intransitive policies for non-deterministic systems. Chong and van der Meyden [22], [23] extend Rushby's model by adopting an epistemic view of the system, i.e., if information flows from H to L via D, then whatever L knows of H must be available to D (in the distributed knowledge of D as a group of downgraders).

In language-based information flow security, Denning et al. [6], [7] first observe that static program analysis can be

used to control information flow in a lattice model. Denning's transitive IFC system is not explicitly connected to noninterference on the semantic level until the work of Volpano and Smith [2], [3] in which Denning's method is formalized as a set of type rules that enforce a simple $L \leq H$ transitive policy. A survey of early development on language-based information flow security can be found in [24].

Strictly transitive noninterference is generally too strong to be usable in realistic programs. Therefore, many approaches to allow controlled releases of information have been devised [25]–[31]. These information declassification or downgrading approaches extend the standard transitive policy with exceptions that grant additional information flow relations once a condition has been fulfilled. For example, DLM [26] provides a transitive IFC framework via multiple ownership of labeled information; the exceptions to this transitive policy are introduced via programmatic constructs of declassification of the information controlled by its owners. DC labels [27] introduce a label format to classify data sensitivities in IFC systems, with privileges that enable downgrading specified as declassification (for confidentiality policies) and endorsement (for integrity policies). Various notions of secure downgrading and declassification have been proposed, differing from each other in respect of their downgrading conditions, as discussed in [20]. Our work presented in this paper is orthogonal to downgrading techniques, as we neither extend any transitive policy nor consider any downgrading condition. However, it may be possible to adopt some downgrading policies to support runtime updates on security labels and information flow relations (e.g. the work on action-state systems [32] and language semantics [33]); this is beyond the scope of this paper.

Rajani and Garg consider the granularity of tracking dependencies in information flow security type systems [9]. They show type systems that track dependencies at the level of individual values are equally expressive to type systems that track dependencies coarsely at the level of entire computation context (which require a construct to limit the scope of the context label), by providing a semantics- and type-preserving translation between them. Their expressiveness means the ability of a type system to type as many semantically secure programs as possible. Their later paper shows a similar result for dynamic IFC systems that additionally allow introspection on labels at run-time [10]. On the other hand, our paper considers the expressiveness of security policy specification with the ability to express as many desired information flow relations as possible. While our security policies can be coarse-grained, our type system is fine-grained and tracks dependencies at the level of individual values. As [9], [10] only consider transitive security policies, it may be interesting to investigate if similar techniques can be adapted to handle nontransitive policies.

Ernst et al. describe a collaborate verification model for high assurance app stores [34], in which app developers provide annotated source code with security policies whose information flow properties are verified by the app store's auditors. Since they focus on detecting Trojan behavior, their security policies only consider information flow in individual apps from system-defined sensitive sources to sensitive sinks. While provided by untrusted developers, their security policies are nontransitive in the sense that transitive flows through source-sink pairs must be explicitly written in the flow policy to prevent indirect flows by apps to whitewash sensitive information. Our work focuses on reasoning about nontransitive information flow among different parts of the system which may each have their own sensitive information. This general approach may be useful for providing constraints on interactions among apps and the system.

VIII. Conclusion

In this paper, we tackle the problem of securing information flow in software composed of untrusted code by proposing a nontransitive IFC system where permissible flow relations cannot be derived from transitivity. Such a system provides a new nontransitive noninterference theorem that can be enforced by a security type system. It ensures that, if a program typechecks, it is information-flow secure, i.e., information flows, either directly or indirectly, are possible only when their flow relations are explicitly specified by the nontransitive policy in effect. This property allows us to reason directly about the extent of information flows in the program, and has potential application in many areas (for example, tracking interactions with untrusted third-party modules, customer-deployed microservices or downloaded mobile apps). This provides a novel contribution to ongoing work investigating the use of type systems, and other formalisms, for securing information flows in programs with complex structures and communication channels. There are fruitful avenues opened up for ongoing research. For us the most promising direction is to investigate incorporating more kinds of security policies, such as possible downgrading properties.

References

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *1982 IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–11.

[2] D. Volpano, C. Irvine, and G. Smith, "A Sound Type System for Secure Flow Analysis," *J. Comput. Secur.*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=353629.353648

[3] D. Volpano and G. Smith, "A type-based approach to program security," in *TAPSOFT '97: Theory and Practice of Software Development*, ser. Lecture Notes in Computer Science, M. Bidoit and M. Dauchet, Eds. Springer Berlin Heidelberg, 1997, pp. 607–621.

[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*, ser. PLAS '09. New York, NY, USA: ACM, 2009, pp. 113–124, event-place: Dublin, Ireland. [Online]. Available: http://doi.acm.org/10.1145/1554339.1554353

[5] ——, "Permissive Dynamic Information Flow Analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:12, event-place: Toronto, Canada. [Online]. Available: http://doi.acm.org/10.1145/1814217.1814220

[6] D. E. Denning, "A Lattice Model of Secure Information Flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976. [Online]. Available: http://doi.acm.org/10.1145/360051.360056

[7] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977. [Online]. Available: http://doi.acm.org/10.1145/359636.359712

[8] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2," in *In Proceedings of the Usenix Symposium on Internet Technologies and Systems*, 1997, pp. 103–112.

[9] V. Rajani and D. Garg, "Types for Information Flow Control: Labeling Granularity and Semantic Models," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, Jul. 2018, pp. 233–246.

[10] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From Fine-to Coarse-grained Dynamic Information Flow Control and Back," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 76:1–76:31, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290389

[11] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," *Commun. ACM*, vol. 17, no. 7, pp. 388–402, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361067

[12] F. Pottier and V. Simonet, "Information Flow Inference for ML," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 1, pp. 117–158, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/596980.596983

[13] M. Pistoia, A. Banerjee, and D. A. Naumann, "Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 149–163.

[14] J. T. Haigh and W. D. Young, "Extending the Noninterference Version of MLS for SAT," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 141–150, Feb. 1987.

[15] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep., dec 1992. [Online]. Available: http://www.csl.sri.com/papers/csl-92-2/

[16] S. Pinsky, "Absorbing covers and intransitive non-interference," in *Proceedings 1995 IEEE Symposium on Security and Privacy*, May 1995, pp. 102–113.

[17] A. W. Roscoe and M. H. Goldsmith, "What is intransitive noninterference?" in *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, Jun. 1999, pp. 228–238.

[18] H. Mantel, "Information Flow Control and Applications - Bridging a Gap -," in *FME 2001: Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, J. N. Oliveira and P. Zave, Eds. Springer Berlin Heidelberg, 2001, pp. 153–172.

[19] H. Mantel and D. Sands, "Controlled Declassification Based on Intransitive Noninterference," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, W.-N. Chin, Ed. Springer Berlin Heidelberg, 2004, pp. 129–145.

[20] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[21] V. Rajani, D. Garg, and T. Rezk, "On access control, capabilities, their equivalence, and confused deputy attacks," in *IEEE 29th Computer Security Foundations Symposium, CSF*. IEEE Computer Society, 2016, pp. 150–163.

[22] R. van der Meyden, "What, indeed, is intransitive noninterference?" *Journal of Computer Security*, vol. 23, no. 2, pp. 197–228, Jan. 2015. [Online]. Available: https://content.iospress.com/articles/journal-of-computer-security/jcs516

[23] S. Chong and R. V. D. Meyden, "Using Architecture to Reason About Information Security," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 2, pp. 8:1–8:30, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2829949

[24] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[25] A. C. Myers, "JFlow: practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, Jan. 1999, pp. 228–241. [Online]. Available: https://doi.org/10.1145/292540.292561

[26] A. C. Myers and B. Liskov, "Protecting Privacy Using the Decentralized Label Model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, Oct. 2000. [Online]. Available: http://doi.acm.org/10.1145/363516.363526

[27] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction Category Labels," in *Information Security Technology for Applications*, ser. Lecture Notes in Computer Science, P. Laud, Ed. Springer Berlin Heidelberg, 2012, pp. 223–239.

[28] L. Waye, P. Buiras, D. King, S. Chong, and A. Russo, "It's My Privilege: Controlling Downgrading in DC-Labels," in *Security and Trust Management*, ser. Lecture Notes in Computer Science, S. Foresti, Ed. Springer International Publishing, 2015, pp. 203–219.

[29] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing Robust Declassification and Qualified Robustness," *Journal of Computer Security*, vol. 14, no. 2, pp. 157–196, Jan. 2006. [Online]. Available: https://content.iospress.com/articles/journal-of-computer-security/jcs258

[30] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable Information Flow Control," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 1875–1891, event-place: Dallas, Texas, USA. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134054

[31] N. Broberg, B. van Delft, and D. Sands, "Paragon for Practical Programming with Information-Flow Control," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, C.-c. Shan, Ed. Springer International Publishing, 2013, pp. 217–232.

[32] S. Eggert and R. van der Meyden, "Dynamic intransitive noninterference revisited," *Formal Aspects of Computing*, vol. 29, no. 6, pp. 1087–1120, Nov. 2017. [Online]. Available: https://doi.org/10.1007/s00165-017-0430-6

[33] B. van Delft, S. Hunt, and D. Sands, "Very Static Enforcement of Dynamic Policies," in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science, R. Focardi and A. Myers, Eds. Springer Berlin Heidelberg, 2015, pp. 32–52.

[34] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative Verification of Information Flow for a High-Assurance App Store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1092–1104, event-place: Scottsdale, Arizona, USA. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660343

APPENDIX

A. Proof of Lemma 6 (Local-Respects)

$$\left.\begin{array}{c}\Gamma, \Pi \vdash_{pc} e : t^{\mathcal{L}} \\ pc \not\sqsubseteq \widehat{\Pi}(\ell) \\ M\ e \Downarrow M'\ v\end{array}\right\} \Longrightarrow M \overset{\widehat{\Pi}(\ell)}{=} M'$$

Proof: As one may observe that in the rules [T-ASN] and [T-NEW], those updated store values are required to be legally influenced by both the program counter pc and the security label of the new value. We show how the equivalence between states can be derived in the following cases which may update the memory M.

- The base case [VAL] is trivial.
- For [ASSIGN], let $M\ e \Downarrow M'\ \iota_\alpha$, $M'\ e' \Downarrow M''\ v$, and $M\ e := e' \Downarrow M''[\iota_\alpha \mapsto v]\ \iota_\alpha$. Also we have $\Gamma\ \Pi \vdash_{pc} e := e' : t^{\mathcal{L}}$. Then by [T-ASN], we have

  $\Gamma\ \Pi \vdash_{pc} e : (\texttt{ref}_{\mathcal{A}}\ t^{\mathcal{L}'})^{\mathcal{L}''}$, (1)

  $\Gamma\ \Pi \vdash_{pc} e' : t^{\mathcal{L}}$, (2)

  $\mathcal{L}'' \subseteq \mathcal{L}$. (3)

  Then by I.H.,

  $M \overset{\widehat{\Pi}(\ell)}{=} M'$ and $M' \overset{\widehat{\Pi}(\ell)}{=} M''$. (4)

  Therefore, by (4) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$,

  $M \overset{\widehat{\Pi}(\ell)}{=} M''$. (5)

  By [T-ASN], $L \cup pc \subseteq \widehat{\Pi}(\alpha)$ for all $\alpha \in \mathcal{A}$. Then we must have $\mathcal{P}(\alpha) \notin \widehat{\Pi}(\ell)$ for all $\alpha \in \mathcal{A}$. Since if $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$, by *Lemma* 4, we would have $pc \subseteq \widehat{\Pi}(\ell)$, which is contradiction.

By *Lemma* 3, $\alpha \in \mathcal{A}$, then $\mathcal{P}(\iota_\alpha) \notin \widehat{\Pi}(\ell)$. Therefore $M'' \overset{\widehat{\Pi}(\ell)}{=} M''[\iota_\alpha \mapsto v]$, which derives $M \overset{\widehat{\Pi}(\ell)}{=} M''[\iota_\alpha \mapsto v]$ by (5) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$, as required.

- For [ALLOC], $M\ e \Downarrow M'\ v$ and $M\ \texttt{new}_\alpha\ e \Downarrow M'[\iota_\alpha \mapsto v]\ \iota_\alpha$. Then by [T-NEW], we have
  $\Gamma\ \Pi \vdash_{pc} \texttt{new}_\alpha\ e : (\texttt{ref}_{\{\alpha\}} t^{\mathcal{L}})^{\mathcal{L}'}$, (6)
  $\Gamma\ \Pi \vdash_{pc} e : t^{\mathcal{L}}$, (7)
  $\mathcal{L} \cup pc \subseteq \widehat{\Pi}(\alpha)$. (8)
  Then by (7) and I.H., $M \overset{\widehat{\Pi}(\ell)}{=} M'$. (9)
  Similar to the above case, by (8), we must have $\mathcal{P}(\alpha) \notin \widehat{\Pi}(\ell)$. Since if $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$, by *Lemma* 4, we would have $pc \subseteq \widehat{\Pi}(\ell)$, which is contradiction. Therefore, $\iota \notin \widehat{\Pi}(\ell)$, and $M' \overset{\widehat{\Pi}(\ell)}{=} M'[\iota_\alpha \mapsto v]$, which derives $M \overset{\widehat{\Pi}(\ell)}{=} M'[\iota_\alpha \mapsto v]$ by transitivity.
- For [THEN], let $M\ e \Downarrow M'\ v$ and $v \neq 0$, and $M'\ e' \Downarrow M''\ v'$. By [T-IF],
  $\Gamma\ \Pi \vdash_{pc} \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' : t^{L}$, (10)
  $\Gamma\ \Pi \vdash_{pc \cup \mathcal{L}'} e'\ t^{\mathcal{L}''}$, (11)
  $\Gamma\ \Pi \vdash_{pc} e\ n^{\mathcal{L}'}$, (12)
  $\mathcal{L} = \mathcal{L}' \cup \mathcal{L}''$. (13)
  By I.H., $pc \not\subseteq \widehat{\Pi}(\ell)$ and (12), we have
  $M \overset{\widehat{\Pi}(\ell)}{=} M'$, (14)
  By $pc \not\subseteq \widehat{\Pi}(\ell)$, we have $pc \cup \mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$, (15)
  By I.H., (11) and (15), we have
  $M' \overset{\widehat{\Pi}(\ell)}{=} M''$. (16)
  Then by (14), (16) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$, we have
  $M \overset{\widehat{\Pi}(\ell)}{=} M''$. (17)
- The [ELSE] case is similar to [THEN].
- For [APP], let $M\ e_1 \Downarrow M'\ \lambda x.e$, $M' \Downarrow M''\ v$, and $M''\ e[v/x] \Downarrow M'''\ v'$. Then by [T-APP],
  $\Gamma\ \Pi \vdash_{pc} e_1\ e_2 : t^{\mathcal{L}}$, (18)
  $\Gamma\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t^{\mathcal{L}_2})^{\mathcal{L}_3}$, (19)
  $\Gamma\ \Pi \vdash_{pc} e_2 : t_1^{\mathcal{L}_1}$, (20)
  $\mathcal{L}_3 \cup pc \subseteq \mathcal{L}'$, (21)
  $\mathcal{L} = \mathcal{L}_2 \cup \mathcal{L}_3$. (22)
  By I.H. and (19), we have $M \overset{\widehat{\Pi}(\ell)}{=} M'$. (23)
  By I.H. and (20), we have $M' \overset{\widehat{\Pi}(\ell)}{=} M''$. (24)
  By (23), (24) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$, we have
  $M \overset{\Pi(\ell)}{=} M''$. (25)
  By [T-FUN], $\Gamma, x : t_1^{\mathcal{L}_1}\ \Pi \vdash_{\mathcal{L}'} e_1 : t^{\mathcal{L}_2}$. (26)
  By *Lemma* 2,
  $\Gamma\ \Pi \vdash_{pc} v : t_1^{\mathcal{L}_1}$, (27)
  then $\Gamma\ \Pi \vdash_{\mathcal{L}'} e_1[v/x] : t^{\mathcal{L}_2}$. (28)
  By $pc \subseteq \mathcal{L}'$ and $pc \not\subseteq \widehat{\Pi}(\ell)$, $\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$. (29)
  Then by I.H. and (29), we have $M'' \overset{\widehat{\Pi}(\ell)}{=} M'''$. (30)
  By (25), (30) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$, we have
  $M \overset{\widehat{\Pi}(\ell)}{=} M'''$. (31)
- All the other cases never update the memory, and the results trivially hold.

□

B. Proof of Lemma 7 (Substitution consistency)

$$\left.\begin{array}{c} M_1 \overset{\widehat{\Pi}(\ell)}{=} M_2 \\ M_1\ e[\vec{u}/\vec{x}] \Downarrow M_1'\ v_1' \\ M_2\ e[\vec{v}/\vec{x}] \Downarrow M_2'\ v_2' \\ \Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e : t_2^{\mathcal{L}} \\ \forall j \in I.\ \mathcal{L}_j \subseteq \widehat{\Pi}(\ell) \Rightarrow u_j = v_j \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} (A)\ M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2' \\ (B)\ \mathcal{L} \subseteq \widehat{\Pi}(\ell) \\ \quad \Rightarrow v_1' = v_2' \end{array}\right.$$

where I is a finite index, and $e[\vec{u}/\vec{x}]$ is short for the substitution $e[u_1/x_1, \ldots u_{|I|}/x_{|I|}]$.

Proof: Suppose $M_1 \overset{\widehat{\Pi}(\ell)}{=} M_2$, we prove by structural induction on $e[\vec{u}/\vec{x}]$ and $e[\vec{v}/\vec{x}]$. For the base case [VAL] there are two cases.

- If both terms are the same value v, trivial.
- If both terms are the same variable x, and the input vectors are singletons u and v with $u = v$, then we have $M_1\ e[u/x] \Downarrow M_1\ u$ and $M_2\ e[v/x] \Downarrow M_2\ v$, and $u = v$.

For the inductive cases, we go through the following rules.

- For [OP], let $M_1\ e_1[\vec{u}/\vec{x}] \oplus e_2[\vec{v}/\vec{x}] \Downarrow M_1''\ c$ and $M_2\ e_1[\vec{u}/\vec{x}] \oplus e_2[\vec{v}/\vec{x}] \Downarrow M_1''\ d$. Then we have $M_1\ e_1[\vec{u}/\vec{x}] \Downarrow M_1'\ c_1$, $M_1'\ e_2[\vec{v}/\vec{x}] \Downarrow M_1''\ c_2$ and $c = c_1 \oplus c_2$. Similarly, $M_2\ e_1[\vec{u}/\vec{x}] \Downarrow M_2'\ d_1$, $M_2'\ e_2[\vec{v}/\vec{x}] \Downarrow M_2''\ d_2$ and $d = d_1 \oplus d_2$. Then by [T-OP],
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_1 \oplus e_2 : n^{\mathcal{L}_1 \cup \mathcal{L}_2}$, (1)
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_1 : n^{\mathcal{L}_1}$, (2)
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_2 : n^{\mathcal{L}_2}$. (3)
  Regarding (A), since all premises are not changed, by I.H., we have
  $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$ and $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$. (4)
  Regarding (B), suppose $\mathcal{L}_1 \cup \mathcal{L}_2 \subseteq \widehat{\Pi}(\ell)$, we have
  $\mathcal{L}_1 \subseteq \widehat{\Pi}(\ell)$ and $\mathcal{L}_2 \subseteq \widehat{\Pi}(\ell)$ (5)
  By I.H., $c_1 = d_1$ and $c_2 = d_2$. (6)
  Given c_1, c_2, d_1, d_2 are all constants, $c = d$. (7)
- For [APP], let $M_1\ e_1[\vec{u}/\vec{x}] \Downarrow M_1'\ \lambda z.e$, $M_1'\ e_2[\vec{u}/\vec{x}] \Downarrow M_1''\ v_1$ and $M_1\ e[\vec{u}/\vec{x}, v_1/z] \Downarrow M_1'''\ v_1'$. Similarly, let $M_2\ e_1[\vec{v}/\vec{x}] \Downarrow M_2'\ \lambda z.e'$, $M_2'\ e_2[\vec{v}/\vec{x}] \Downarrow M_2''\ v_2$ and $M_2\ e'[\vec{v}/\vec{x}, v_2/z] \Downarrow M_2'''\ v_2'$. By [T-APP], $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_1\ e_2 : t^{\mathcal{L}}$, $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t_2^{\mathcal{L}_2})^{\mathcal{L}_3}$, $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i \in I}\ \Pi \vdash_{pc} e_2 : t_1^{\mathcal{L}_1}$, $\mathcal{L}_3 \cup pc \subseteq \mathcal{L}'$ and $\mathcal{L} = \mathcal{L}_2 \cup \mathcal{L}_3$.

  Suppose $\mathcal{L} \subseteq \widehat{\Pi}(\ell)$**.** Then $\mathcal{L}_3 \subseteq \widehat{\Pi}(\ell)$. We need to show both (A) and (B).

  Then by I.H., $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$, (8)
  and $\lambda z.e = \lambda z.e'$ (9)
  We have the following two cases.
  - Suppose $\mathcal{L}_1 \subseteq \widehat{\Pi}(\ell)$. (10)
    By I.H. and (10), $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$ and $v_1 = v_2$. (11)

By *Lemma* 2, we have
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} v_1 : t_1^{\mathcal{L}_1}$
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} v_2 : t_1^{\mathcal{L}_1}$.
Given $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t_2^{\mathcal{L}_2})^{\mathcal{L}_3}$, then by [T-FUN],
we have $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}, \{z : t_1^{\mathcal{L}_1}\} \vdash_{\mathcal{L}'}\ e : t_2^{\mathcal{L}_2}$(12)
By $\mathcal{L}_2 \subseteq \mathcal{L}$ and (10), we have $\mathcal{L}_2 \subseteq \widehat{\Pi}(\ell)$ (13)
By (12), (13) and I.H., we have
$M_1''' \overset{\widehat{\Pi}(\ell)}{=} M_2'''$ and $v_1' = v_2'$. (14)

– Suppose $\mathcal{L}_1 \not\subseteq \widehat{\Pi}(\ell)$. (15)

By I.H., we have $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$. (16)
(but it is possible that $v_1 \neq v_2$)
By *Lemma* 2,
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} v_1 : t_1^{\mathcal{L}_1}$, (17)
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} v_2 : t_1^{\mathcal{L}_1}$. (18)
Given $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t_2^{\mathcal{L}_2})^{\mathcal{L}_3}$, by [T-FUN], we have
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}, \{z : t_1^{\mathcal{L}_1}\} \vdash_{\mathcal{L}'}\ e : t_2^{\mathcal{L}_2}$. (19)
We define a new index $I' = I \cup \{|I|+1\}$, and let $x_{|I|+1} = z$ and
$t_{|I|+1}^{\mathcal{L}_{|I|+1}} = t_1^{\mathcal{L}_1}$. By $\mathcal{L}_1 \not\subseteq \widehat{\Pi}(\ell)$, the new input vector $\{x_j : t_j^{\mathcal{L}_j}\}_{I'}$ satisfies for all $j \in I'$:
$(\mathcal{L}_j \subseteq \ell) \Rightarrow u_j = v_j$. (20)
By (13), (17), (18), (20) and I.H., we have
$M_1''' \overset{\widehat{\Pi}(\ell)}{=} M_2'''$ and $v_1' = v_2'$. (21)

Suppose $\mathcal{L} \not\subseteq \widehat{\Pi}(\ell)$**,** we only need to show proof obligation (A). We have the following two cases.

– Suppose $\mathcal{L}_3 \not\subseteq \widehat{\Pi}(\ell)$. (22)

By I.H., we still have $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$. (23)
(but it is possible that $\lambda z.e \neq \lambda z.e'$)

Similarly, by I.H., we have $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$. (24)

Given $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t_2^{\mathcal{L}_2})^{\mathcal{L}_3}$, by [T-FUN], we have $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}, \{z : t_1^{\mathcal{L}_1}\} \vdash_{\mathcal{L}'} e : t_2^{\mathcal{L}_2}$.

By $\mathcal{L}_3 \subseteq \mathcal{L}'$, we have $\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$. (25)
Then by *Lemma* 6, we have
$M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_1'''$ and $M_2'' \overset{\widehat{\Pi}(\ell)}{=} M_2'''$. (26)
Then by (24), (26) and transitivity of $\overset{\widehat{\Pi}(\ell)}{=}$, we have
$M_1''' \overset{\widehat{\Pi}(\ell)}{=} M_2'''$. (27)

– Suppose $\mathcal{L}_3 \subseteq \ell$ and $\mathcal{L}_2 \not\subseteq \widehat{\Pi}(\ell)$. (28)

By I.H., $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$ and $\lambda z.e = \lambda z.e'$. (29)

Similarly, by I.H., we have $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$ (30)
and $\mathcal{L}_1 \subseteq \widehat{\Pi}(\ell)$ implies $v_1 = v_2$. (31)

Given $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e_1 : (t_1^{\mathcal{L}_1} \xrightarrow{\mathcal{L}'} t_2^{\mathcal{L}_2})^{\mathcal{L}_3}$, by [T-FUN],
we have $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}, \{z : t_1^{\mathcal{L}_1}\} \vdash_{\mathcal{L}'}\ e : t_2^{\mathcal{L}_2}$.
We define a new index $I' = I \cup \{|I|+1\}$, and let
$x_{|I|+1} = z$ and $t_{|I|+1}^{\mathcal{L}_{|I|+1}} = t_1^{\mathcal{L}_1}$. (32)
By (31), no matter $\mathcal{L}_1 \subseteq \widehat{\Pi}(\ell)$ or $\mathcal{L}_1 \not\subseteq \widehat{\Pi}(\ell)$,
the new input vector $\{x_j : t_j^{\mathcal{L}_j}\}_{I'}$ satisfies for all
$j \in I' : (\mathcal{L}_j \subseteq \ell) \Rightarrow u_j = v_j$. (33)

by (33) and I.H., we have $M_1''' \overset{\widehat{\Pi}(\ell)}{=} M_2'''$. (34)

- For [ASSIGN], let $M_1\ e_1[\vec{u}/\vec{x}] \Downarrow M_1'\ \iota_1$, $M_1'\ e_2[\vec{u}/\vec{x}] \Downarrow M_1''\ v_1$ and $M_1\ e_1[\vec{u}/\vec{x}] := e_2[\vec{u}/\vec{x}] \Downarrow M_1''[\iota_1 \mapsto v_1]\ v_1$. Similarly, $M_2\ e_1[\vec{v}/\vec{x}] \Downarrow M_2'\ \iota_2$, $M_2'\ e_2[\vec{v}/\vec{x}] \Downarrow M_2''\ v_2$ and $M_2\ e_1[\vec{v}/\vec{x}] := e_2[\vec{v}/\vec{x}] \Downarrow M_2''[\iota_2 \mapsto v_2]\ v_2$.
By [T-ASN],
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e_1 := e_2 : t^{\mathcal{L}'}$, (35)
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I} \vdash_{pc} e_1 : (\mathtt{ref}_{\mathcal{A}} t^{\mathcal{L}})^{\mathcal{L}'}$ (36)
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I} \vdash_{pc} e_2 : t^{\mathcal{L}'}$, (37)
$\forall \alpha \in \mathcal{A}.\mathcal{L}' \cup pc \trianglerighteq \mathcal{P}(\alpha)$ and $\mathcal{L} \cup \mathcal{L}' \cup pc \subseteq \widehat{\Pi}(\alpha)$. (38)
We have the following two cases.

– Suppose $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$. (39)
By (36) and I.H., we have
$M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$ and $\iota_1 = \iota_2$. (40)
By (37) and I.H., we have
$M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$ and $v_1 = v_2$. (41)
By (40) and (41), we have
$M_1''[\iota_1 \mapsto v_1] \overset{\widehat{\Pi}(\ell)}{=} M_2''[\iota_2 \mapsto v_2]$, and $v_1 = v_2$. (42)

– Suppose $\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$. (43)

By (36) and I.H., $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$. (44)

By (37) and I.H., $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$. (45)
Since $\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$, we must have for all $\alpha \in \mathcal{A}$: $\mathcal{P}(\alpha) \notin \widehat{\Pi}(\ell)$. Because if $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$, by [T-ASN], $\mathcal{L}' \subseteq \widehat{\Pi}(\alpha)$, then by *Lemma* 4, we would have $L' \subseteq \widehat{\Pi}(\ell)$, which is contradiction. By *Lemma* 3, both ι_1 and ι_2 correspond to a program counter in $\mathcal{A}$. Therefore $\mathcal{P}(\iota_1) \notin \widehat{\Pi}(\ell)$ and $\mathcal{P}(\iota_2) \notin \widehat{\Pi}(\ell)$, then by (45), we have $M_1''[\iota_1 \mapsto v_1] \overset{\widehat{\Pi}(\ell)}{=} M_2''[\iota_2 \mapsto v_2]$.

- For [THEN] and [ELSE], let $M_1\ e[\vec{u}/\vec{x}] \Downarrow M_1'\ v_1$. W.l.o.g., let $v_1 \neq 0$, and $M_1'\ e'[\vec{u}/\vec{x}] \Downarrow M_1''\ v_1'$, which derives $M\ \mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ e'' \Downarrow M_1''\ v_1'$.
By [T-IF],
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} \mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ e'' : t^{\mathcal{L}}$,(46)
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc\cup\mathcal{L}'} e'\ t^{\mathcal{L}''}$, (47)
$\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e\ n^{\mathcal{L}'}$, (48)
and $\mathcal{L} = \mathcal{L}' \cup \mathcal{L}''$. (49)
We have the following three cases.

– If $\mathcal{L} \subseteq \widehat{\Pi}(\ell)$, we have $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$, then by (48) and I.H., $M_2\ e[\vec{v}/\vec{x}] \Downarrow M_2'\ v_2$, $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$ and $v_1 = v_2 \neq 0$. Since $\mathcal{L}'' \subseteq \mathcal{L}$, we have $\mathcal{L}'' \subseteq \ell$, then again by (47) and I.H., $M_2'\ e'[\vec{v}/\vec{x}] \Downarrow M_2''\ v_2'$, $v_1' = v_2'$ and $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$.

– If $\mathcal{L} \not\subseteq \widehat{\Pi}(\ell)$ and $\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$, then by (48) and I.H., $M_2\ e \Downarrow M_2'\ v_2$ and $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$. Note it is possible that $v_1 \neq v_2$. W.l.o.g., let $v_2 = 0$. Since

$\mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$, we have $pc \cup \mathcal{L}' \not\subseteq \widehat{\Pi}(\ell)$, then by *Lemma* 6, $M_2'\ e''[\vec{v}/\vec{x}] \Downarrow M_2''\ v_2'$ and $M_2' \overset{\widehat{\Pi}(\ell)}{=} M_2''$. Similarly, by *Lemma* 6, $M_1'\ e'[\vec{u}/\vec{x}] \Downarrow M_1''\ v_1'$ and $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_1''$. Therefore $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$ by transitivity of the relation $\overset{\widehat{\Pi}(\ell)}{=}$.

- If $\mathcal{L} \not\subseteq \widehat{\Pi}(\ell)$, $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$ and $\mathcal{L}'' \not\subseteq \widehat{\Pi}(\ell)$. Since $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$, by (48) and I.H., $M_2\ e[\vec{v}/\vec{x}] \Downarrow M_2'\ v_2$, $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$ and $v_1 = v_2 \neq 0$. Therefore both executions choose the first branch e'. Although $\mathcal{L}'' \not\subseteq \widehat{\Pi}(\ell)$, by (47) and I.H., we still have $M_2'\ e'[\vec{v}/\vec{x}] \Downarrow M_2''\ v_2'$ and $M_1'' \overset{\widehat{\Pi}(\ell)}{=} M_2''$.

Suppose $v_1 = 0$, this is the [ELSE] case which can be treated in a way symmetric to the [THEN] case.

- For [ALLOC], let $M_1\ e[\vec{u}/\vec{x}] \Downarrow M_1'\ v_1$ and $M_1'\ \texttt{new}_\alpha\ e[\vec{u}/\vec{x}] \Downarrow M_1'[\iota_1 \mapsto v_1]\ \iota_1$. Also let $M_2\ e[\vec{v}/\vec{x}] \Downarrow M_2'\ v_2$ and $M_2'\ \texttt{new}_\alpha\ e[\vec{v}/\vec{x}] \Downarrow M_2'[\iota_2 \mapsto v_2]\ \iota_2$.

  By [T-NEW],
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} \texttt{new}_\alpha\ e : (\texttt{ref}_{\{\alpha\}} t^{\mathcal{L}})^{\mathcal{L}'}$ (50)
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e : t^{\mathcal{L}}$, (51)
  $\mathcal{L} \cup pc \subseteq \widehat{\Pi}(\alpha)$, (52)
  $\mathcal{P}(\alpha) \in \mathcal{L}'$. (53)

  By (51) and I.H., we have $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$. (54)

  Regarding (A), we have the following two cases.
  - If $\mathcal{L} \subseteq \widehat{\Pi}(\ell)$, then $v_1 = v_2$ by I.H.(B). Then whether or not $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$, we always have $M_1'[\iota_1 \mapsto v_1] \overset{\widehat{\Pi}(\ell)}{=} M_2'[\iota_2 \mapsto v_2]$.
  - If $\mathcal{L} \not\subseteq \widehat{\Pi}(\ell)$, we must have $\mathcal{P}(\alpha) \notin \widehat{\Pi}(\ell)$. Because if $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$, by (52) and *Lemma* 4, we would have $\mathcal{L} \subseteq \widehat{\Pi}(\ell)$, contradiction. Since $\mathcal{P}(\alpha) \notin \widehat{\Pi}(\ell)$, we have $M_1'[\iota_1 \mapsto v_1] \overset{\widehat{\Pi}(\ell)}{=} M_2'[\iota_2 \mapsto v_2]$.

  Regarding (B), suppose $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$. By (53), $\mathcal{P}(\alpha) \in \widehat{\Pi}(\ell)$. Then by *Lemma* 1 and (54), we have $\iota_1 = \iota_2$.

- For [DEREF], let $M_1\ e[\vec{u}/\vec{x}] \Downarrow M_1'\ \iota_1$, $M'(\iota_1) = v_1$ and $M_1\ !e[\vec{u}/\vec{x}] \Downarrow M_1'\ v_1$. Similarly, let $M_2\ e[\vec{u}/\vec{x}] \Downarrow M_2'\ \iota_2$, $M'(\iota_2) = v_2$ and $M_2\ !e[\vec{u}/\vec{x}] \Downarrow M_2'\ v_2$.

  By [T-DEREF],
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} e : (\texttt{ref}_{\mathcal{A}} t^{\mathcal{L}})^{\mathcal{L}'}$, (55)
  $\Gamma, \{x_i : t_i^{\mathcal{L}_i}\}_{i\in I}\ \Pi \vdash_{pc} !e : t^{\mathcal{L}\cup\mathcal{L}'}$, (56)
  $\forall \alpha \in \mathcal{A}.\ \widehat{\Pi}(\alpha) \subseteq \mathcal{L} \cup \mathcal{L}'$. (57)

  Then by (55) and I.H., we have $M_1' \overset{\widehat{\Pi}(\ell)}{=} M_2'$. (58)
  If $\mathcal{L} \cup \mathcal{L}' \subseteq \widehat{\Pi}(\ell)$, then $\mathcal{L}' \subseteq \widehat{\Pi}(\ell)$, and by I.H., we have $\iota_1 = \iota_2$. (59)
  By *Lemma* 2, $\Gamma, \Pi \vdash_{pc} M_1$ and $\Gamma, \Pi \vdash_{pc} M_2$, we have
  $\Gamma, \Pi \vdash_{pc} M_1'$ and $\Gamma, \Pi \vdash_{pc} M_2'$ (60)
  By [MEMORY] and (60), we have
  $\Gamma, \Pi \vdash_{pc} M_1'(\iota_1) : \Pi(\alpha)$ and (61)
  $\Gamma, \Pi \vdash_{pc} M_2'(\iota_2) : \Pi(\alpha)$. (62)
  By (57), we have $\widehat{\Pi}(\alpha) \subseteq \widehat{\Pi}(\ell)$. (63)
  By (58), (59) and (63), we have $M_1'(\iota_1) = M_2'(\iota_2)$,
  i.e., $v_1 = v_2$. (64)

□