Nontransitive Policies Transpiled

Citation for the original published paper (version of record):
Ahmadpanah, S., Askarov, A., Sabelfeld, A. (2021). Nontransitive Policies Transpiled. Proceedings - 2021 IEEE European Symposium on Security and Privacy, Euro S and P 2021: 543-561.
http://dx.doi.org/10.1109/EuroSP51992.2021.00043

Nontransitive Policies Transpiled

Mohammad M. Ahmadpanah[*], Aslan Askarov[†], and Andrei Sabelfeld[*]

[*]*Chalmers University of Technology*
[†]*Aarhus University*

***Abstract*—Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) are a new security condition and enforcement for policies which, in contrast to Denning's classical lattice model, assume no transitivity of the underlying flow relation. Nontransitive security policies are a natural fit for coarse-grained information-flow control where labels are specified at module rather than variable level of granularity.**

While the nontransitive and transitive policies pursue different goals and have different intuitions, this paper demonstrates that nontransitive noninterference can in fact be reduced to classical transitive noninterference. We develop a lattice encoding that establishes a precise relation between NTNI and classical noninterference. Our results make it possible to clearly position the new NTNI characterization with respect to the large body of work on noninterference. Further, we devise a lightweight program transformation that leverages standard flow-sensitive information-flow analyses to enforce nontransitive policies. We demonstrate several immediate benefits of our approach, both theoretical and practical. First, we improve the permissiveness over (while retaining the soundness of) the nonstandard NTT enforcement. Second, our results naturally generalize to a language with intermediate inputs and outputs. Finally, we demonstrate the practical benefits by utilizing state-of-the-art flow-sensitive tool JOANA to enforce nontransitive policies for Java programs.

I. Introduction

Modern approaches to secure information flow follow Denning's classical model [8]. This model maps information to security levels and uses a flow relation that regulates how information can move between the levels. Under Denning's model, when data moves from one security level to another one, it effectively looses its original security classification. Denning therefore argues that in such a model, the flow relation must be transitive, which has been the convention for a large body of work on information flow control [32], [26], [13].

Nontransitive policies In recent work, Lu and Zhang [17] observe that in certain scenarios, the transitivity requirement is in fact undesirable. This is most apparent when security policies are specified in a coarse-grained manner, i.e., at the level of mutually-distrustful components in an application. For example, "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" [17]. Another, more fine-grained example, is that of user policies in a social network stipulating that "my friends can access my personal data but not friends of my friends". To semantically characterize such security requirements, Lu and Zhang propose the notion of *nontransitive* noninterference (NTNI) and propose a specially designed type system to statically enforce it.

Nontransitive noninterference is not to be confused with *intransitive* noninterference [25], [23], [18], [30], a popular model for declassification. Although both nontransitive and intransitive policies assume flow relations are not transitive, there is a conceptual difference between them. Assuming a flow relation with flows from A to B and from B to C but not from A to C, intransitive noninterference allows A's information to indirectly flow to C as long as the information passes through a declassifier. In contrast, nontransitive policy forbids all flows from A to C. Section VII elaborates the relation in detail.

NTNI is introduced by a nonstandard security characterization and a specialized type system [17]. The question remains open whether the mainstream machinery of information-flow control reasoning and enforcement can be leveraged for tracking NTNI.

This paper answers this question positively by showing how to encode nontransitive noninterference via classical transitive noninterference. Our encoding makes it possible to use standard transitive techniques for information-flow control to enforce nontransitive policies and thus address the coarse-grained scenarios that motivate them. This has substantial practical benefits, making it possible to deploy information-flow concepts and tools to achieve nontransitive security.

We argue that flow-sensitive analysis is a natural fit for the component-based scenario, where developers are not required to provide fine-grained annotations at the level of variables. We devise a lightweight program transformation to leverage flow-sensitive information-flow analysis to enforce NTNI. Thanks to the flow-sensitivity of the analysis, the type system verifies which variables are affected by what components, enforcing component-level security. We implement a prototype of the transpiler, i.e., program transformer and policy translator, and leverage flow-sensitive static tool JOANA [11] to demonstrate our approach in practice.

Contributions The contributions of this paper are:

- We show that the definition of NTNI can be reduced to classical transitive noninterference through a lattice encoding (Section II).
- We leverage our encoding to show how an existing flow-sensitive information-flow type system can enforce the coarse-grained policies that motivate NTNI in the first place (Section III).

- We extend our results to a language supporting interaction through input and output commands (Section IV).
- We develop a prototype that translates NTNI policy to a classical transitive setting and uses JOANA static analysis tool (Section V).

II. Security characterization transpiled

All permitted flows between security levels are expressed explicitly under nontransitive policies, as opposed to the traditional way [8] of policy specification where security levels constitute a partially ordered set. Nontransitive policies only have reflexive property, yet expressive enough to include other properties such as transitivity and antisymmetry among arbitrary selections of levels.

This section shows how nontransitive noninterference can be modeled as transitive noninterference using a power-lattice encoding. Throughout the paper, we use a running example adopted from Lu and Zhang [17] to discuss how the transpilation works. We formalize the security notions and prove the relation between these two approaches to define a security policy.

Running example Figure 1 shows our running example consisting of three components named `Alice`, `Bob`, and `Charlie`. The security policy stipulates that `Bob` is allowed to read `Alice`'s information and `Charlie` is allowed to read `Bob`'s information. At the same time, no information flow from `Alice` is allowed to `Charlie`.

Based on the policy, `Bob` can only send information to `Charlie` if it is not influenced by `Alice`, as illustrated in Figure 2. A transitive policy would presume that if information may flow from `Alice` to `Bob` and from `Bob` to `Charlie`, then it may also flow from `Alice` to `Charlie`. This is not the case in this example. Since nontransitive policies specify all permitted flows explicitly, the information flow from `Alice` to `Charlie` would be considered as desired only if it was explicitly stated in the policy. It is indeed easy to see that nontransitive policies are a generalization of transitive ones because transitive closures can be stated as permitted flows to preserve the transitive property.

```
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; }
}
```

Figure 1: Running example [17].

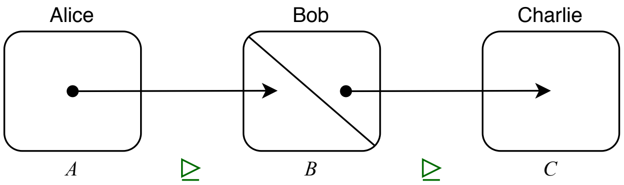


Figure 2: Nontransitive policy for the running example.

Using a coarse-grained information-flow control is sufficient to specify the intended policy. Consider the labels A, B, and C for the components `Alice`, `Bob`, and `Charlie`, respectively. We specify the nontransitive policy using an arbitrary information flow relation $\trianglerighteq$[1], written $A \trianglerighteq B$ and $B \trianglerighteq C$, which specifies that information from security level A can flow to security level B and from B to C. It also stipulates any other information flows between the levels are disallowed. For instance, information from security level A must not flow to C, directly or through any other components.

For the sake of simplicity, we rewrite the example program in a model language (without support for object-orientation) that demonstrates the explicit flows arisen from data dependencies between component variables. In the program shown in Figure 3, `Comp.var` denotes the variable `var` belongs to the component `Comp`.

```
// Bob.receive(data)
Bob.data1 := Alice.data;
// Bob.good()
Charlie.data := Bob.data2;
// Bob.bad()
Charlie.data := Bob.data1;
```

Figure 3: Simplified version of the running example.

To track flows between component variables, we label all variables of a component with the security label of the component. By extending the labeling function for variables of components, we classify `Alice.data` as A, `Bob.data1` and `Bob.data2` as B, and `Charlie.data` as C. The program does not satisfy nontransitive noninterference because there is an illegal flow from A to C; the content of `Alice.data` is directly transmitted to `Charlie.data` via `Bob.data1`. If the program, however, did not include the `bad` method in `Bob`, it would be secure with respect to the nontransitive policy.

A. *Security notions*

We now present our model language and formal definitions of security notions, i.e., transitive and nontransitive noninterference for programs. To model the essence of these characterizations, we assume a simple batch-job setting where only the initial and final memories are observable (before and after program execution). We will show how to extend our results to a language with I/O in Section IV.

[1] As a visual cue, we will use the green color for nontransitive and blue color for transitive notions throughout the paper.

$$\begin{aligned}
e &::= v \mid x \mid e \oplus e\\
c &::= \mathit{skip} \mid x := e \mid \mathit{if}\ e\ \mathit{then}\ c\ \mathit{else}\ c \mid \mathit{while}\ e\ \mathit{do}\ c \mid c; c
\end{aligned}$$

Figure 4: Language syntax.

Expression Evaluation

$$\frac{}{\langle v, M\rangle \Downarrow v} \qquad (\textsc{Value})$$

$$\frac{}{\langle x, M\rangle \Downarrow M(x)} \qquad (\textsc{Read})$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \qquad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2} \qquad (\textsc{Operation})$$

Command Evaluation

$$\frac{}{\langle \mathit{skip}, M\rangle \to \langle \mathit{stop}, M\rangle} \qquad (\textsc{Skip})$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad M' = M[x \mapsto v]}{\langle x := e, M\rangle \to \langle \mathit{stop}, M'\rangle} \qquad (\textsc{Write})$$

$$\frac{c = \mathit{if}\ e\ \mathit{then}\ c_{\mathit{true}}\ \mathit{else}\ c_{\mathit{false}} \qquad \langle e, M\rangle \Downarrow b}{\langle c, M\rangle \to \langle c_b, M\rangle} \qquad (\textsc{If})$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{true}}{\langle c, M\rangle \to \langle c_{\mathit{body}}; c, M\rangle} \qquad (\textsc{While-T})$$

$$\frac{c = \mathit{while}\ e\ \mathit{do}\ c_{\mathit{body}} \qquad \langle e, M\rangle \Downarrow \mathit{false}}{\langle c, M\rangle \to \langle \mathit{stop}, M\rangle} \qquad (\textsc{While-F})$$

$$\frac{\langle c_1, M\rangle \to \langle c_1', M'\rangle}{\langle c_1; c_2, M\rangle \to \langle c_1'; c_2, M'\rangle} \qquad (\textsc{Seq-I})$$

$$\frac{}{\langle \mathit{stop}; c, M\rangle \to \langle c, M\rangle} \qquad (\textsc{Seq-II})$$

Figure 5: Language semantics.

Programs consist of multiple code components and a memory $M : \mathit{Var} \to \mathit{Val}$, a (total) mapping from a set of variables Var to a set of values Val, partitioned by components Cmp of the program. A variable $x_\alpha \in \mathit{Var}$ denotes x is allocated at $\alpha \in \mathit{Cmp}$. We write x where the component name is unused. Using coarse-grained labeling, each component maps to a security label, written $\Gamma_{\mathit{Cmp}} : \mathit{Cmp} \to L$. As a result, all variables of a component are annotated with the same label. Formally, $\forall \alpha \in \mathit{Cmp}. \forall x_\alpha \in \mathit{Var}. \Gamma(x_\alpha) = \Gamma_{\mathit{Cmp}}(\alpha)$ where $\Gamma : \mathit{Var} \to L$. Note that we use Var_c for the set of variables that exist in program c.

Figures 4 and 5 illustrate the syntax and semantics of our model language. An execution configuration $\langle c, M\rangle$ is a pair of a command c and a given memory M, and $\to$ introduces the transition relation between configurations. For expressions, $\langle e, M\rangle \Downarrow v$ denotes an expression e evaluates to a value v under a memory M. We write $\to^*$ for the reflexive and transitive closure of the $\to$ relation, and $\to^n$ for the n-step execution of $\to$.

We adopt *termination-insensitive* [32] noninterference that ignores information leaks resulted from termination behavior of the given program. NTNI is introduced by a termination-insensitive notion for batch-job programs [17]. We extend the model language to support I/O and lift the security notion to progress-insensitive [3].

Note that the choices of termination- and progress-sensitivity are orthogonal to nontransitivenesses of policies. Our results (in particular, the lattice encoding) can be thus replayed for other variants of noninterference.

Transitive Noninterference (TNI) For a given program, classical noninterference guarantees if two memories agree on variables at level ℓ and lower, memories after the execution of the program also agree on the variables at level ℓ and lower. Accordingly, an observer at level ℓ can see the values of the variables labeled as ℓ or lower, called *ℓ-observable* values. Transitive noninterference stipulates ℓ-observable final values of a program only depend on initial values from ℓ or lower levels.

A transitive security policy is a triple $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ where $L_{\mathcal{T}}$ is a set of security labels and $\sqsubseteq \subseteq L_{\mathcal{T}} \times L_{\mathcal{T}}$ is a binary relation that forms a partially ordered set (reflexivity, asymmetry, transitivity) on $L_{\mathcal{T}}$ and specifies permitted flows between security levels. A labeling function $\Gamma_{\mathcal{T}} : \mathit{Var} \to L_{\mathcal{T}}$ maps a variable to a security label.

Transitive indistinguishability relation ($=_{\mathcal{T}}$) for a security label $\ell \in L_{\mathcal{T}}$ is defined as follows. Two memories are indistinguishable at level ℓ if and only if values of variables observable at the level ℓ and lower are the same.

Definition 1 (*Transitive Memory Indistinguishability*)**.** *Two memories M_1 and M_2 are transitively indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $M_1 \overset{\ell}{=}_{\mathcal{T}} M_2$ if and only if $\forall x \in \mathit{Var}. \Gamma_{\mathcal{T}}(x) \sqsubseteq \ell \implies M_1(x) = M_2(x)$.*

We define transitive noninterference based on the indistinguishability relation between memories. A (batch-job) program c satisfies *termination-insensitive transitive noninterference*, written $\mathit{TNI}_{\mathit{TI}}(\mathcal{T}, c)$, when for any two memories indistinguishable at level $\ell \in L_{\mathcal{T}}$, the computation of the program c terminates for both and the ℓ-observer cannot distinguish the final memories.

Definition 2 (*Termination-Insensitive Transitive Noninterference*)**.** *A program c satisfies $\mathit{TNI}_{\mathit{TI}}(\mathcal{T}, c)$ if and only if $\forall \ell \in L_{\mathcal{T}}. \forall M_1, M_2. \big(M_1 \overset{\ell}{=}_{\mathcal{T}} M_2 \wedge \langle c, M_1\rangle \to^* \langle \mathit{stop}, M_1'\rangle \wedge \langle c, M_2\rangle \to^* \langle \mathit{stop}, M_2'\rangle\big) \implies M_1' \overset{\ell}{=}_{\mathcal{T}} M_2'$.*

Nontransitive Noninterference (NTNI) The nontransitive notion of noninterference demands that for a given program, changes on variables at security level ℓ can only influence variables at the levels allowed by the policy. In this condition, *ℓ-observable* values are the content of variables labeled as ℓ. Hence, nontransitive noninterference ensures that ℓ-observable final values are only dependent on those initial values that *can flow* to ℓ, as stated in the policy.

A nontransitive security policy is a triple $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}}$ is a set of security labels, $\Gamma_{\mathcal{N}} \colon Var \rightarrow L_{\mathcal{N}}$ is a labeling function, and $\trianglerighteq$ is an arbitrary flow relation specifying permitted flows (*can-flow-to* relation [8]). We define $C(\ell) = \{\ell' | \ell' \trianglerighteq \ell\}$ as the set of levels that can flow to ℓ, including itself. The only condition for the relation is to be reflexive; no other properties, such as transitivity, are required.

Nontransitive indistinguishability relations ($=_{\mathcal{N}}$) for a security label $\ell \in L_{\mathcal{N}}$ and a set of security labels $\mathcal{L} \subseteq L_{\mathcal{N}}$ are defined below. Two memories are indistinguishable at level ℓ if variables of the level ℓ have the same values in those two. Consistently, the relation holds for a set of labels if variables of any level existing in the set be mapped to same values in the two memories.

Definition 3 (*Nontransitive Memory Indistinguishability*)**.** *Two memories M_1 and M_2 are nontransitively indistinguishable at level $\ell \in L_{\mathcal{N}}$, written $M_1 \overset{\ell}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in Var. \Gamma_{\mathcal{N}}(x) = \ell \implies M_1(x) = M_2(x)$. The memories are indistinguishable for a set of security levels $\mathcal{L} \subseteq L_{\mathcal{N}}$, written $M_1 \overset{\mathcal{L}}{=}_{\mathcal{N}} M_2$, if and only if $\forall x \in Var. \Gamma_{\mathcal{N}}(x) \in \mathcal{L} \implies M_1(x) = M_2(x)$.*

We use the indistinguishability relation between memories to define nontransitive noninterference. A (batch-job) program c satisfies *termination-insensitive nontransitive noninterference*, written $NTNI_{TI}(\mathcal{N}, c)$, if for any two memories indistinguishable for the set of levels may influence variables at $\ell \in L_{\mathcal{N}}$, the program c gets terminated for both and the ℓ-observer cannot distinguish the final memories.

Definition 4 (*Termination-Insensitive Nontransitive Noninterference*)**.** *A program c satisfies $NTNI_{TI}(\mathcal{N}, c)$ if and only if $\forall \ell \in L_{\mathcal{N}}. \forall M_1, M_2. \big(M_1 \overset{C(\ell)}{=}_{\mathcal{N}} M_2 \wedge \langle c, M_1 \rangle \rightarrow^* \langle stop, M_1' \rangle \wedge \langle c, M_2 \rangle \rightarrow^* \langle stop, M_2' \rangle\big) \implies M_1' \overset{\ell}{=}_{\mathcal{N}} M_2'$.*

B. Relationship between NTNI and TNI

We first prove that NTNI is a generalization of TNI, and then for the other side, we introduce the transpilation from NTNI to TNI and discuss how a nontransitive policy can be seen as transitive. We present an encoding to convert nontransitive policies to transitive ones and show if a program is secure with respect to a nontransitive policy, then a semantically equivalent program satisfies an equivalent transitive policy and vice versa.

Theorem 1 (*From TNI_{TI} to $NTNI_{TI}$*)**.** For any program c and any transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, there exists a nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}$, $\trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$ such that $TNI_{TI}(\mathcal{T}, c) \iff NTNI_{TI}(\mathcal{N}, c)$. Formally,

$$\forall c. \forall \mathcal{T}. \exists \mathcal{N}. TNI_{TI}(\mathcal{T}, c) \iff NTNI_{TI}(\mathcal{N}, c).$$

Proof. To conserve space, the proofs of all statements can be found in the online appendix in the supplementary materials [1]. □

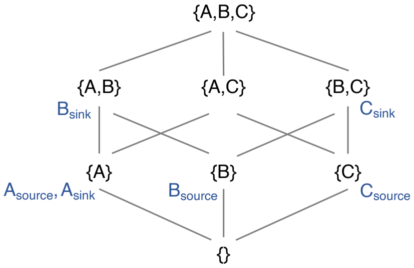


Figure 6: The powerset lattice for the running example.

The transpilation from NTNI to TNI includes mapping the nontransitive policy to the corresponding transitive one and rewriting the given program to be compatible with the policy encoding. We establish a powerset lattice with the set of security levels. To connect these two policies together, we should map the components and their variables to the transitive labels. Prior to labeling variables, a transformation in the program is needed, which we call *canonicalization*.

In nontransitive policies, $A \trianglerighteq B$ means information from the source level A can flow to the sink level B. Therefore, we allocate two fresh variables for each component variable to capture the source and sink of information. We prepend a sequence of assignments from source variables to the component variables, and we append assignments from the component variables to sink variables. Then, we can label source and sink variables separately with respect to the encoding to preserve the notion of nontransitive policy.

Running example We describe the transpilation from NTNI to TNI for the running example shown in Figure 3. We form the powerset lattice of labels used in the nontransitive policy as the set of labels for the corresponding transitive policy, i.e., $L_{\mathcal{T}} = \wp(\{A, B, C\})$ and $\sqsubseteq = \subseteq$ (see Figure 6). We transform the program to be able to capture the notion of nontransitive noninterference by assigning labels to variables. We add two fresh variables for each component variable in the given program to differentiate the source and sink of information and label them according to the definition of NTNI.

Figure 7 demonstrates the program after the transformation, which we call it the *canonical* version of the program. It consists of three sections: (1) initial assignments from a (source) variable to a `temp` variable (lines 2-5), (2) a copy of the program where variables are replaced by `temp` variables (lines 7-9), and (3) final assignments from `temp` to `sink` variables (lines 12-15). It is obvious that the meaning of the program is preserved in the transformation.

Next, we define the new labeling function for component variables. As illuminated by annotations in Figure 6, for any component variable `Comp.x` that the component `Comp` is labeled as ℓ in nontransitive policy, we label (source) variables `Comp.x` as $\{\ell\}$, `Comp.x_temp` as the top element of the security lattice, i.e., the set of all nontransitive labels, and `Comp.x_sink` as the set of nontransitive labels that can flow to the variable, i.e., $C(\ell)$. Thus, information flows from source variables (labeled $\{\ell\}$) to sink variables (labeled $C(\ell)$) are carried through internal *temp* variables. In Section III, we show how the

```
1   // init
2   Alice.data_temp := Alice.data;
3   Bob.data1_temp := Bob.data1;
4   Bob.data2_temp := Bob.data2;
5   Charlie.data_temp := Charlie.data;
6
7   Bob.data1_temp := Alice.data_temp;
8   Charlie.data_temp := Bob.data2_temp;
9   Charlie.data_temp := Bob.data1_temp;
10
11  // final
12  Alice.data_sink := Alice.data_temp;
13  Bob.data1_sink := Bob.data1_temp;
14  Bob.data2_sink := Bob.data2_temp;
15  Charlie.data_sink := Charlie.data_temp;
```

Figure 7: Canonical version of the running example.

presented type system updates the type of *temp* variables based on data and control flows and verifies whether the final assignments are secure.

Having the described labeling function, the canonical version of the given program does not satisfy the transitive policy. By tracking the sequence of lines 2, 7, 9, and 15 in Figure 7, an explicit flow from $\{A\}$ (level of `Alice.data`) to $\{B, C\}$ (level of `Charlie.data_sink`) is identified, which is not permitted with respect to the transitive policy ($\{A\} \nsubseteq \{B, C\}$). However, similar to the original program and the nontransitive policy, if the program did not include the undesired flow, the program would be considered secure.

Program canonicalization Algorithm 1 explains the transformation for batch-job programs. First, for each variable x in the program, we allocate two fresh variables $x_{temp}, x_{sink} \in Var \setminus Var_c$, and then apply the following transformation on the given program. We use ++ to denote the operator for string concatenation and the notation $c\,[x \mapsto x_{temp}]$ indicates renaming all occurrences of x in program c to x_{temp} (in a capture-avoiding manner). We use Var_{temp} and Var_{sink} to point to the set of *temp* and *sink* variables, respectively.

Algorithm 1: Canonicalization algorithm for batch-job programs.

```
Input  : Program $c$
Output: Program $Canonical(c)$
$init$ := “”
$final$ := “”
foreach $x \in Var_c$ do
    $c\,[x \mapsto x_{temp}]$
    $init := init$ ++ “$x_{temp} := x$;”
    $final := final$ ++ “; $x_{sink} := x_{temp}$”
end
$Canonical(c) := init$ ++ $c$ ++ $final$
return $Canonical(c)$
```

We prove that the canonical version of the program keeps the meaning and termination behavior of the original program, yet the final values of variables are in the *sink* variables.

Lemma 1 (*Semantic Equivalence Modulo Canonicalization*)**.** For any program c, the semantic equivalence $\simeq_C$ between the programs c and $Canonical(c)$ holds, where $c \simeq_C c' \overset{def}{=} \forall M.\big(\langle c, M\rangle \rightarrow^* \langle \text{stop}, M'\rangle \iff \langle c', M\rangle \rightarrow^* \langle \text{stop}, M''\rangle\big) \wedge \forall x \in Var_c.\big(M'(x) = M''(x_{temp}) = M''(x_{sink}) \wedge M(x) = M''(x)\big)$.

The following lemmas are intermediate steps to show how a nontransitive policy on a given program is reduced to a transitive policy using the powerset lattice resulted from the set of nontransitive labels in combination with the canonical version of the program. Lemma 2 proves that the transformation holds a program secure with respect to a nontransitive policy if and only if the original program is secure.

Lemma 2 (*$NTNI_{TI}$ Preservation under Canonicalization*)**.** Any program c is secure with respect to a nontransitive security policy $\mathcal{N}$ if and only if the canonical program $Canonical(c)$ is secure where $\forall x \in Var_c.\Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$. Formally,

$$\forall c.\forall \mathcal{N}.NTNI_{TI}(\mathcal{N}, c) \iff NTNI_{TI}(\mathcal{N}, Canonical(c)).$$

We define the powerset encoding of a nontransitive policy to a transitive policy for canonical programs as follows.

Definition 5 (*Transitive Encoding of Nontransitive Policies*)**.** *Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ and a program c, the corresponding transitive policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ on the canonical version of the program is $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$, $\sqsubseteq\, = \subseteq$, and*

$$\forall x \in Var_c.\begin{cases}\Gamma_{\mathcal{T}}(x) & = & \{\Gamma_{\mathcal{N}}(x)\}\\ \Gamma_{\mathcal{T}}(x_{temp}) & = & L_{\mathcal{N}}\\ \Gamma_{\mathcal{T}}(x_{sink}) & = & C(\Gamma_{\mathcal{N}}(x))\end{cases}.$$

As stated in Definition 5, the initial and final values of an ℓ-observable variable x of the given program are $\{\ell\}$- and $C(\ell)$-observable in the canonical version, respectively. Also, *temp* variables are internal and the top-level observer only can see their final values, thus $L_{\mathcal{N}}$-observable. The next lemma demonstrates for any canonical program satisfying a nontransitive policy, the program also complies with a corresponding transitive policy and vice versa.

Lemma 3 (*From $NTNI_{TI}$ to TNI_{TI} for Canonical Programs*)**.** Any canonical program $Canonical(c)$ is secure with respect to a nontransitive security policy $\mathcal{N}$ where $\forall x \in Var_c.\Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$ if and only if the canonical program is secure according to the corresponding transitive security policy $\mathcal{T}$. We write $\forall c.\forall \mathcal{N}.\exists \mathcal{T}.\ NTNI_{TI}(\mathcal{N}, Canonical(c)) \iff TNI_{TI}(\mathcal{T}, Canonical(c))$.

Finally, by connecting the previous lemmas, we prove that any nontransitive policy on a given program can be modeled as a transitive policy on the canonical version of the program. Given Theorems 1 and 2, the two notions of *transitive* and *nontransitive* noninterference coincide.

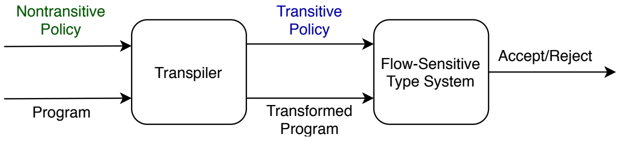


Figure 8: Composition of transpiler and enforcement mechanism.

Theorem 2 (*From $NTNI_{TI}$ to TNI_{TI}*)**.** For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}} \rangle$, there exist a semantically equivalent (modulo canonicaliztion) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}} \rangle$, as specified in Definition 5, such that $NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c')$. Formally,

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_C c' \wedge NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c').$$

III. Enforcement transpiled

The proposed enforcement mechanism for nontransitive policies [17] is a type system that does not use subtyping, the classical way to check transitive types, for information flow verification. Instead, it tracks dependencies between program variables and collects all security labels of flows into a component variable throughout the program. Then it checks whether the flows comply with the specified policy. Therefore, the type system can enforce both nontransitive and transitive policies.

To enforce a nontransitive policy, however, we can benefit from the transpilation introduced in Section II and devise a transitive type system for canonical programs. We employ a (vanilla) flow-sensitive type system [14] enforcing the corresponding transitive policy on transformed programs. The flow-sensitivite type system investigates how components influence variables of the program. Figure 8 illustrates the composition of the transpiler and the enforcement mechanism.

We prove soundness of our transitive type system (Figure 9a) and investigate how it relates to the nontransitive type system. Inspired by the notion, we present a nontransitive type system for our model language (Figure 9b) and prove the soundness property. Then, we show that the flow-sensitive transitive type system accepts more secure programs compared to the nontransitive one.

A. Enforcement mechanism

We present a flow-sensitive type system that enforces transitive policies for canonical programs. The type system allows updates of security types through typing the program. When an expression is assigned to a variable, the security type of the variable changes to the join of security types of the expression and the program counter, to capture explicit and implicit flows (arisen from control dependencies) to the variable.

For a command c, judgments are in the form of $pc \vdash \Gamma\{c\}\Gamma'$, where $pc \in L_{\mathcal{T}}$ is the program counter label and the typing environment $\Gamma : Var \to L_{\mathcal{T}}$ will be updated to Γ' after execution of c. We make use of the structure of canonical programs in the typing rules, presented in Figure 9a. The two rules for assignments (rules TT-WRITE-I and TT-WRITE-II) represent the essence of the type system. We know that only *temp* and *sink* variables can be on the left-hand side of an assignment in a canonical program. Assignments to *sink* variables occur at the end of the program, i.e., the final section, where the right-hand side of assignments are *temp* variables (rule TT-WRITE-II). The type system allows changes to the security types, except for *sink* variables, whose initial types must be kept (rule TT-SUB). Otherwise, upgrading security levels of *sink* variables might violate the soundness property of the type system.

Running example Given the policy specified in the running example, the type system rejects the canonical program shown in Figure 7. The initial types of the variables are the sets of labels introduced in Definition 5. Applying the typing rules, the types of the variables `Alice.data_temp`, `Bob.data1_temp`, and `Charlie.data_temp` are (at least) the same as the type of `Alice.data`, which is $\{A\}$. The assignments in the final section are well-typed except for the last one, where the type of `Charlie.data_sink` is the set of labels can flow to C, i.e., $\{B, C\}$. Since $\{A\} \nsubseteq \{B, C\}$, the program is ill-typed with respect to the given nontransitive policy. We will discuss more examples in Section V.

The next theorem states soundness of the flow-sensitive type system, which means if the type system accepts a canonical program, then the program satisfies the transitive noninterference, and consequently, the original program complies with the nontransitive policy.

Theorem 3 (*Soundness of Flow-Sensitive Transitive Type System*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Canonical(c)\}\Gamma' \;\Rightarrow\; TNI_{TI}(\mathcal{T}, Canonical(c)).$$

B. Relationship between nontransitive and flow-sensitive transitive type systems

The core idea of Lu and Zhang's type system [17] is tracking data and control dependencies between program variables through type inference on information propagation history. Then it guarantees flow relations from inferred labels of dependencies to the specified label of the variable are stated in the policy. Their flow-insensitive type system captures all possible dependencies to a variable; thus it becomes less permissive in comparison with a flow-sensitive type system. Given the semantic relationship between nontransitive and transitive policies, we demonstrate our flow-sensitive transitive type system accepts all the well-typed programs in the nontransitive type system, and more secure programs.

We present a nontransitive type system for our imperative model language based on the essence of their type system. It aggregates security labels of data and control dependencies of variables through the program. For each assignment $x := e$, the type system checks permission of information flows from the collected labels of the expression e and the program counter to the specified label of the variable x.

Typing judgments are in the form of $\mathcal{P}, \Gamma, pc \vdash c : t$ that indicates the type t is assigned to the command c with respect to the program counter label $pc \subseteq L_{\mathcal{N}}$ in the typing

$\dfrac{}{\Gamma \vdash v : \bot}$	(TT-VALUE)	$\dfrac{}{\Gamma \vdash v : \varnothing}$	(NT-VALUE)
$\dfrac{}{\Gamma \vdash x : \Gamma(x)}$	(TT-READ)	$\dfrac{}{\Gamma \vdash x : \Gamma(x)}$	(NT-READ)
$\dfrac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2}$	(TT-OPERATION)	$\dfrac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \cup t_2}$	(NT-OPERATION)
$\dfrac{}{pc \vdash \Gamma\{skip\}\,\Gamma}$	(TT-SKIP)	$\dfrac{}{\mathcal{P}, \Gamma, pc \vdash skip : t}$	(NT-SKIP)
$\dfrac{\Gamma \vdash e : t \qquad x \in Var_{temp}}{pc \vdash \Gamma\{x := e\}\,\Gamma[x \mapsto pc \sqcup t]}$	(TT-WRITE-I)	$\dfrac{\begin{array}{c}\Gamma \vdash e : t \qquad \Gamma \vdash x : t \\ \forall \ell \in t \cup pc.\ell \in \Gamma(x) \wedge \ell \trianglerighteq \mathcal{P}(x)\end{array}}{\mathcal{P}, \Gamma, pc \vdash x := e : t}$	(NT-WRITE)
$\dfrac{\begin{array}{c}x' \in Var_{temp} \qquad x \in Var_{sink} \\ pc \sqcup \Gamma(x') \sqsubseteq \Gamma(x)\end{array}}{pc \vdash \Gamma\{x := x'\}\,\Gamma}$	(TT-WRITE-II)		
$\dfrac{\begin{array}{c}\Gamma \vdash e : t \\ pc \sqcup t \vdash \Gamma\{c_{true}\}\,\Gamma' \\ pc \sqcup t \vdash \Gamma\{c_{false}\}\,\Gamma'\end{array}}{pc \vdash \Gamma\{if\ e\ then\ c_{true}\ else\ c_{false}\}\,\Gamma'}$	(TT-IF)	$\dfrac{\begin{array}{c}\Gamma \vdash e : t_1 \\ \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{true} : t_2 \\ \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{false} : t_2\end{array}}{\mathcal{P}, \Gamma, pc \vdash if\ e\ then\ c_{true}\ else\ c_{false} : t_1 \cup t_2}$	(NT-IF)
$\dfrac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_{body}\}\,\Gamma}{pc \vdash \Gamma\{while\ e\ do\ c_{body}\}\,\Gamma}$	(TT-WHILE)	$\dfrac{\Gamma \vdash e : t_1 \qquad \mathcal{P}, \Gamma, pc \cup t_1 \vdash c_{body} : t_2}{\mathcal{P}, \Gamma, pc \vdash while\ e\ do\ c_{body} : t_1 \cup t_2}$	(NT-WHILE)
$\dfrac{pc \vdash \Gamma\{c_1\}\,\Gamma' \qquad pc \vdash \Gamma'\{c_2\}\,\Gamma''}{pc \vdash \Gamma\{c_1; c_2\}\,\Gamma''}$	(TT-SEQ)	$\dfrac{\mathcal{P}, \Gamma, pc \vdash c_1 : t_1 \qquad \mathcal{P}, \Gamma, pc \vdash c_2 : t_2}{\mathcal{P}, \Gamma, pc \vdash c_1; c_2 : t_1 \cup t_2}$	(NT-SEQ)
$\dfrac{\begin{array}{c}pc_1 \vdash \Gamma_1\{c\}\,\Gamma'_1 \\ pc_2 \sqsubseteq pc_1 \qquad \Gamma_2 \sqsubseteq \Gamma_1 \qquad \Gamma'_1 \sqsubseteq \Gamma'_2 \\ \forall x \in Var_{sink}.\Gamma_1(x) = \Gamma_2(x) = \Gamma'_1(x) = \Gamma'_2(x)\end{array}}{pc_2 \vdash \Gamma_2\{c\}\,\Gamma'_2}$	(TT-SUB)	$\dfrac{\Gamma \vdash e : t_1 \qquad t_1 \subseteq t_2}{\Gamma \vdash e : t_2}$	(NT-SUB-I)
		$\dfrac{\begin{array}{c}\mathcal{P}, \Gamma, pc_1 \vdash c : t_1 \\ pc_2 \subseteq pc_1 \qquad t_1 \subseteq t_2\end{array}}{\mathcal{P}, \Gamma, pc_2 \vdash c : t_2}$	(NT-SUB-II)
(a)		(b)	

Figure 9: (a) Transitive typing rules; (b) Nontransitive typing rules.

environments $\mathcal{P} : Var \to L_{\mathcal{N}}$ and $\Gamma : Var \to \wp(L_{\mathcal{N}})$ where $\forall x \in Var.\mathcal{P}(x) \in \Gamma(x)$. Figure 9b illustrates the typing rules where $\mathcal{P}$ specifies the nontransitive levels of variables and Γ predicts the set of labels that might influence the final value of a variable in the program.

The most important rule is the one for typing assignments (rule NT-WRITE). The set $\Gamma(x)$ must contain all possible information flows to the variable x in the program, which is checked in the premise $(t \cup pc \subseteq \Gamma(x))$, and then the type system verifies whether those are permitted flows or not $(\forall \ell \in t \cup pc.\ell \trianglerighteq \mathcal{P}(x))$. Note that the specified label of a variable $\mathcal{P}(x)$ must be present in the set of dependencies $\Gamma(x)$ because the $\trianglerighteq$ relation is reflexive.

Running example The nontransitive type system tracks and collects all the security labels a variable has a dependency on through the program and checks whether they are compliant with the permitted flows. Therefore, the program presented in Figure 3 is rejected by the type system because the type of `Bob.data1` must be (at least) $\{A, B\}$ to record the type of `Alice.data`, which is $\{A\}$ and $A \trianglerighteq B$ exists in the policy. Consequently, the last assignment is ill-typed respecting the typing environment and absence of $A \trianglerighteq C$ in the policy.

In the following, we prove soundness of the nontransitive type system. Any well-typed program with respect to the nontransitive typing rules satisfies nontransitive noninterference.

Theorem 4 (*Soundness of Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma, pc \vdash c : t \Rightarrow NTNI_{TI}(\mathcal{N}, c).$$

On closer inspection, both type systems are sound but the nontransitive type system is not as permissive as the flow-sensitive mechanism. The flow-sensitive transitive type system updates the labels of variables based on the flow of the program in a more precise manner. The next theorem shows if a program is secure under the nontransitive type system, the flow-sensitive type system accepts the canonical version of the program as well.

Theorem 5 (*Flow-Sensitive Type System Covers Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma_1, pc \vdash c : t \Rightarrow pc \vdash \Gamma_2\{Canonical(c)\}\,\Gamma_3,$$

where $\forall x \in Var_c.\Gamma_3(x_{temp}) \sqsubseteq \Gamma_1(x) \wedge \Gamma_2(x) = \Gamma_3(x) = \{\mathcal{P}(x)\} \wedge \Gamma_2(x_{temp}) = L_{\mathcal{N}} \wedge \Gamma_2(x_{sink}) = \Gamma_3(x_{sink}) = C(\mathcal{P}(x))$.

The counterexample program in Figure 10 demonstrates the theorem does not hold in the other direction; there is a well-typed program according to the flow-sensitive rules, which gets rejected by the nontransitive type system. If we swap the last two statements of the running example, as shown in Figure 10, the nontransitive type system still rejects the program; types of both sides of an assignment must be the same (rule NT-WRITE). The flow-sensitive type system, however, accepts the program because it detects that the last assignment overwrites the final value of `Charlie.data` and updates the label accordingly (rule TT-WRITE-I). It can be shown that adding flow-sensitivity flavor to the nontransitive type system enhances precision to the same level offered by the flow-sensitive transitive type system.

```
// Bob.receive(data)
Bob.data1 := Alice.data;
// Bob.bad()
Charlie.data := Bob.data1;
// Bob.good()
Charlie.data := Bob.data2;
```

Figure 10: An example that shows the flow-sensitive type system is more permissive than the nontransitive type system.

IV. EXTENSION WITH I/O

We extend the model language to support input and output commands. In this setting, sources and sinks of information are more tangible, as a better fit for real-world programs with third-party components. Interestingly, we will observe a more natural correspondence between nontransitive and transitive security notions.

A. Security notions

Programs can receive inputs and produce outputs at any step of computation. We include two new constructs $input(x,\ell)$ and $output(x,\ell)$ for reading a value from the input channel at security level ℓ and sending a value to the output channel at level ℓ, respectively. This model entails a revision on security notions where intermediate output values are observable as well as the termination behavior of a program.

We naturally choose another notion of noninterference named *progress-insensitive* [3], [13] (corresponding to *CP-security* for reactive programs [5]) that demands if two program inputs agree on values at security levels may influence variables at ℓ, the output sequence observable at level ℓ remains the same up to the point that one of the executions diverges silently (without producing any output). Transitive policies define an input/output value *ℓ-observable* if the value is at level ℓ or lower, while an ℓ-observer in a nontransitive policy only sees values at level ℓ. Note that the termination behavior of a program is observable for all security levels in both security notions.

Running example Recall the nontransitive policy of the running example in Section II: $A \trianglerighteq B$ and $B \trianglerighteq C$. The program in Figure 11 violates progress-insensitive nontransitive noninterference due to the presence of an implicit flow from the input value of `Alice.data` with security level A to the observable output at level C. Based on the input value, the program sends an output value at level B or C. Therefore, the observable outputs are different at levels B and C, depending on the input value at level A.

```
input(Alice.data, A);
Bob.data1 := Alice.data;
if Bob.data1 then
    output(Bob.data2, B);
else
    output(Charlie.data, C);
```

Figure 11: Running example with I/O.

Figure 12 illustrates the syntax of our model language supporting I/O. Evalution rules for input and output commands are presented in Figure 13. We refer to Figure 23 (in Appendix) for the complete set of semantic rules. An execution configuration $\langle c,M,I,O\rangle$ is a tuple consists of a command c, a memory M, an input function I that maps security levels to input channels, and an output channel O. The relation $\rightarrow$ defines transitions between configurations. We assume the environment is input total. We model program inputs as a mapping from security levels to sequences of values, written $I(\ell) = v.\sigma$, where $\ell \in L$, $v \in Val$, and σ is a sequence of values. We define output behavior of a program recursively by $O = \varnothing \mid \circlearrowleft \mid v_\ell.O$, where $\circlearrowleft$ denotes silent divergence. Based on the language semantics, we abstract away details of computation steps and define output evaluation of an execution. Definition 6 introduces the new relation $\rightsquigarrow$ that indicates an initial configuration $\langle c,M,I,\varnothing\rangle$ evaluates to O.

Definition 6 (*Output Behavior of A Program Execution*)**.** *The output behavior O generated by an initial execution configuration $\langle c,M,I,\varnothing\rangle$, written $\langle c,M,I,\varnothing\rangle \rightsquigarrow O$, is defined as follows:*

$$\frac{\langle c,M,I,\varnothing\rangle \rightarrow^* \langle stop,M',I',O\rangle}{\langle c,M,I,\varnothing\rangle \rightsquigarrow O}$$

$$\frac{\begin{array}{c}\langle c,M,I,\varnothing\rangle \rightarrow^* \langle c',M',I',O\rangle \\ \forall n \in \mathbb{N}.\, \langle c',M',I',O\rangle \rightarrow^n \langle c_n,M_n,I_n,O\rangle \wedge c_n \neq stop\end{array}}{\langle c,M,I,\varnothing\rangle \rightsquigarrow O.\circlearrowleft}.$$

Transitive Noninterference (TNI) Classical noninterference guarantees ℓ-observable output behavior of a program only depends on inputs from ℓ or lower levels. A transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ is a triple where $L_{\mathcal{T}}$ is a set of security labels and $\sqsubseteq\, \subseteq L_{\mathcal{T}} \times L_{\mathcal{T}}$ is a binary relation that specifies permitted flows between security levels forming a partially ordered set on $L_{\mathcal{T}}$. A labeling function $\Gamma_{\mathcal{T}} : Var \rightarrow L_{\mathcal{T}}$ maps a variable to a security label.

The definition of progress-insensitive noninterference relies on the definition of indistinguishability relations for inputs and outputs. To define the relations, we should first describe observable inputs and outputs at a security level ℓ. An ℓ-observer can see the content of input channels at the security

$$
\begin{aligned}
e &::= v \mid x \mid e \oplus e\\
c &::= \mathit{skip} \mid x := e \mid \mathit{if}\ e\ \mathit{then}\ c\ \mathit{else}\ c \mid \mathit{while}\ e\ \mathit{do}\ c \mid c;c \mid\\
&\quad\ \mathit{input}(x,\ell) \mid \mathit{output}(x,\ell)
\end{aligned}
$$

Figure 12: Language syntax with I/O.

$$
\frac{\begin{array}{cc} c = \mathit{input}(x,\ell) & I(\ell) = v.\sigma \\ I' = I[\ell \mapsto \sigma] & M' = M[x \mapsto v] \end{array}}{\langle c, M, I, O\rangle \to \langle \mathit{stop}, M', I', O\rangle}\ (\text{IO-INPUT})
$$

$$
\frac{\begin{array}{cc} \multicolumn{2}{c}{c = \mathit{output}(x,\ell)} \\ M(x) = v & O' = O.\,v_{\ell} \end{array}}{\langle c, M, I, O\rangle \to \langle \mathit{stop}, M, I, O'\rangle}\ (\text{IO-OUTPUT})
$$

Figure 13: Language semantics with I/O (selected rules).

level ℓ and lower. We define observable output behavior at a level $\ell \in L_{\mathcal{T}}$ by purging the values from an output sequence which are not at the level ℓ or lower.

Definition 7 (*Transitive Observable Output Behavior*)**.** *Given an output behavior O including a sequence of output values and termination behavior of a program execution. The subsequence of the output behavior observable at a security level $\ell \in L_{\mathcal{T}}$ is defined below:*

$$
O|_{\ell}^{\mathcal{T}} = \begin{cases} O & O = \varnothing \lor O = \circlearrowright \\ v_{\ell'}.\,O'|_{\ell}^{\mathcal{T}} & O = v_{\ell'}.\,O' \land \ell' \sqsubseteq \ell \\ O'|_{\ell}^{\mathcal{T}} & \mathit{otherwise} \end{cases} .
$$

We call two program inputs indistinguishable at level $\ell \in L_{\mathcal{T}}$ if input sequences of the levels ℓ are the same as well as lower levels.

Definition 8 (*Transitive Input Indistinguishability*)**.** *Two program inputs I_1 and I_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $I_1 \overset{\ell}{=}_{\mathcal{T}} I_2$, if and only if $\forall \ell' \sqsubseteq \ell.\, I_1(\ell') = I_2(\ell')$.*

Two program outputs are indistinguishable at level ℓ when the sequences of observable outputs are exactly the same up to the silent divergence in one of them. In other words, if both of the output behaviors are terminating, then the ℓ-observable subsequences must be identical. Otherwise, the subsequences must be the same until one of them reaches the $\circlearrowright$ event.

Definition 9 (*Transitive Output Indistinguishability*)**.** *Two program outputs O_1 and O_2 are indistinguishable at level $\ell \in L_{\mathcal{T}}$, written $O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$, if and only if $O_1|_{\ell}^{\mathcal{T}} = O_2|_{\ell}^{\mathcal{T}} \lor (\exists O, O'.\, O_1|_{\ell}^{\mathcal{T}} = O.\circlearrowright \land O_2|_{\ell}^{\mathcal{T}} = O.\,O') \lor (\exists O, O'.\, O_1|_{\ell}^{\mathcal{T}} = O.\,O' \land O_2|_{\ell}^{\mathcal{T}} = O.\circlearrowright)$.*

Given the indistinguishability definitions, we are ready to define the security condition. A program c satisfies *progress-insensitive transitive noninterference*, written $TNI_{PI}(\mathcal{T}, c)$, when for any two program inputs indistinguishable at level $\ell \in L_{\mathcal{T}}$, the output behaviors resulted from the execution of the program are indistinguishable for the ℓ-observer.

Definition 10 (*Progress-Insensitive Transitive Noninterference*)**.** *A program c satisfies $TNI_{PI}(\mathcal{T}, c)$ if and only if $\forall \ell \in L_{\mathcal{T}}.\,\forall M.\,\forall I_1, I_2.\, I_1 \overset{\ell}{=}_{\mathcal{T}} I_2 \land \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1 \Rightarrow \exists O_2.\langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \land O_1 \overset{\ell}{=}_{\mathcal{T}} O_2$.*

Nontransitive Noninterference (NTNI) The nontransitive notion of noninterference stipulates that ℓ-observable output behavior of a given program is only dependent on those inputs that *can flow* to ℓ, as stated in the policy. A nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ is a triple where $L_{\mathcal{N}}$ is a set of security labels, $\trianglerighteq$ is an arbitrary flow relation specifying permitted flows, and $\Gamma_{\mathcal{N}} \colon \mathit{Var} \to L_{\mathcal{N}}$ is a labeling function.

Similar to the transitive notion, we define indistinguishability relations for program inputs and outputs with respect to definitions of observable inputs and outputs at a security level, respectively. An ℓ-observer can see the content of the input channel at the level ℓ and the subsequence of output values at the level ℓ as well as the divergence event.

Definition 11 (*Nontransitive Observable Output Behavior*)**.** *Given an output behavior O including a sequence of output values and termination behavior of a program execution. The subsequence of the output behavior observable at a security level $\ell \in L_{\mathcal{N}}$ is defined as follows:*

$$
O|_{\ell}^{\mathcal{N}} = \begin{cases} O & O = \varnothing \lor O = \circlearrowright \\ v_{\ell}.\,O'|_{\ell}^{\mathcal{N}} & O = v_{\ell}.\,O' \\ O'|_{\ell}^{\mathcal{N}} & \mathit{otherwise} \end{cases} .
$$

Two program inputs are indistinguishable for a set of levels $\mathcal{L} \subseteq L_{\mathcal{N}}$ if input sequences of the levels member of $\mathcal{L}$ are identical with each other.

Definition 12 (*Nontransitive Input Indistinguishability*)**.** *Two program inputs I_1 and I_2 are indistinguishable for a set of levels $\mathcal{L} \subseteq L_{\mathcal{N}}$, written $I_1 \overset{\mathcal{L}}{=}_{\mathcal{N}} I_2$, if and only if $\forall \ell \in \mathcal{L}.\, I_1(\ell) = I_2(\ell)$.*

Similar to Definition 9, two program outputs are indistinguishable at level $\ell \in L_{\mathcal{N}}$ if the sequences of observable outputs are the same until one of the executions diverges silently.

Definition 13 (*Nontransitive Output Indistinguishability*)**.** *Two program outputs O_1 and O_2 are indistinguishable at level $\ell \in L_{\mathcal{N}}$, written $O_1 \overset{\ell}{=}_{\mathcal{N}} O_2$, if and only if $O_1|_{\ell}^{\mathcal{N}} = O_2|_{\ell}^{\mathcal{N}} \lor (\exists O, O'.\, O_1|_{\ell}^{\mathcal{N}} = O.\circlearrowright \land O_2|_{\ell}^{\mathcal{N}} = O.\,O') \lor (\exists O, O'.\, O_1|_{\ell}^{\mathcal{N}} = O.\,O' \land O_2|_{\ell}^{\mathcal{N}} = O.\circlearrowright)$.*

Having the indistinguishability relations in hand, we define the noninterference notion for the nontransitive setting. A program c satisfies *progress-insensitive nontrasnitive noninterference*, written $NTNI_{PI}(\mathcal{N}, c)$, when for any two program inputs indistinguishable for the set of levels may influence variables at level $\ell \in L_{\mathcal{N}}$, the output behaviors resulted from the execution of the program are indistinguishable for the ℓ-observer.

Definition 14 (*Progress-Insensitive Nontransitive Noninterference*)**.** *A program c satisfies $NTNI_{PI}(\mathcal{N}, c)$ if and only if* $\forall \ell \in L_{\mathcal{N}}. \forall M. \forall I_1, I_2. I_1 \overset{C(\ell)}{=}_{\mathcal{N}} I_2 \wedge \langle c, M, I_1, \varnothing\rangle \rightsquigarrow O_1 \Rightarrow \exists O_2. \langle c, M, I_2, \varnothing\rangle \rightsquigarrow O_2 \wedge O_1 \overset{\ell}{=}_{\mathcal{N}} O_2$.

B. Relationship between NTNI and TNI

We follow the same pattern to relate nontransitive and transitive security definitions together. Constructing the power-lattice encoding remains as before, although the transformation algorithm is more straightforward for programs with input/outputs. Before we see that, the next theorem confirms NTNI is still a generalization of TNI using the progress-insensitive notion in the security definitions.

Theorem 6 (*From TNI_{PI} to $NTNI_{PI}$*)**.** For any program c and any transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$, there exists a nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ where $L_{\mathcal{N}} = L_{\mathcal{T}}, \trianglerighteq = \sqsubseteq^*$, and $\Gamma_{\mathcal{N}} = \Gamma_{\mathcal{T}}$ such that $TNI_{PI}(\mathcal{T}, c) \iff NTNI_{PI}(\mathcal{N}, c)$. Formally,

$$\forall c. \forall \mathcal{T}. \exists \mathcal{N}. TNI_{PI}(\mathcal{T}, c) \iff NTNI_{PI}(\mathcal{N}, c).$$

We introduce the transpilation for programs with intermediate input/outputs. Similar to the batch-job style, we establish the powerset lattice out of nontransitive labels, i.e., $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$ and $\sqsubseteq = \subseteq$. However, the transformation algorithm is quite simpler than canonicalization; only input and output commands are required to be rewritten because of the new security definition that considers only the relation between program inputs and outputs.

Program transformation As explained in Algorithm 2, we label sources of information at a security level $\ell \in L_{\mathcal{N}}$ as the singleton set of a security level ($\{\ell\}$) and annotate sinks as the set of labels that can flow to ℓ, or $C(\ell)$. More precisely, we replace *input*(x, ℓ) commands with *input*$(x, \{\ell\})$, and also *output*(x, ℓ) commands with *output*$(x, C(\ell))$ in the program.

Algorithm 2: Transformation algorithm for programs with I/O.

```
Input  : Program $c$
Output: Program $Transform(c)$
foreach $x \in Var_c$ do
    $c\,[input(x, \ell) \mapsto input(x, \{\ell\})]$
    $c\,[output(x, \ell) \mapsto output(x, C(\ell))]$
end
$Transform(c) := c$
return $Transform(c)$
```

Running example Figure 14 demonstrates how the transformation works on the running example. Each output command explicitly specifies the set of labels that are permitted to influence the output value. The transformed program does not satisfy transitive noninterference because the presence of output value at level $\{B, C\}$ depends on an input value at level $\{A\}$, which are incomparable in the security lattice. However, the flow from the input value to the output value at level $\{A, B\}$ is permitted because $\{A\} \subseteq \{A, B\}$.

```
input(Alice.data, {A});
Bob.data1 := Alice.data;
if Bob.data1 then
    output(Bob.data2, {A,B});
else
    output(Charlie.data, {B,C});
```

Figure 14: Transformed version of running example with I/O.

It is obvious that the transformed version of a given program preserves the meaning and termination behavior of the original program, yet it changes the channel of output values. The input and output values at the level ℓ can be found on the input channel with label $\{\ell\}$ and the output channel labeled as $C(\ell)$ in the canonical version of the given program. The next lemma shows the semantic relation between a given program and the transformed one.

Lemma 4 (*Semantic Equivalence Modulo Transformation*)**.** For any program c, the semantic equivalence $\simeq_T$ between the programs c and *Transform*(c) holds where $c \simeq_T c' \overset{def}{=} \forall M. \forall I. \exists I'. \big(\forall \ell. I(\ell) = I'(\{\ell\})\big) \wedge \langle c, M, I, \varnothing\rangle \rightsquigarrow O \wedge \langle c', M, I', \varnothing\rangle \rightsquigarrow O' \wedge O' = O\,[v_\ell \mapsto v_{C(\ell)}]$.

Then, we prove a nontransitive policy on a given program (with intermediate inputs/outputs) can be reduced to a transitive policy on the transformed version of the program. Theorems 6 and 7 demonstrate the mutual relationship between NTNI and TNI holds, even for programs with intermediate observable values.

Theorem 7 (*From $NTNI_{PI}$ to TNI_{PI}*)**.** For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo transformation) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ where $c' =$ *Transform*(c), $L_{\mathcal{T}} = \wp(L_{\mathcal{N}})$, $\sqsubseteq = \subseteq$ and $\forall x \in Var_c. \Gamma_{\mathcal{T}}(x) = \{\Gamma_{\mathcal{N}}(x)\}$ such that $NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c')$. Formally,

$$\forall \mathcal{N}. \forall c. \exists \mathcal{T}. \exists c'. c \simeq_T c' \wedge NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c').$$

C. Enforcement mechanism

Figure 15 illustrates an excerpt from a flow-sensitive type system enforcing transitive policies on transformed programs. We refer to Figure 24 (in Appendix) for the complete set of typing rules. The type system defines judgments of the form $pc \vdash \Gamma\{c\}\Gamma'$ where $pc \in L_{\mathcal{T}}$ is the program counter label, and the typing environments $\Gamma : Var \rightarrow L_{\mathcal{T}}$ and Γ' describe the security levels of variables before and after executing the command c, respectively. Security types of the variables get updated freely through the program and capture the information flows to the variable (rule IO-TT-WRITE).

The rules for typing input and output commands are the most important ones. The typing environments before and after executing an output command stay the same if the explicit flows ($\Gamma(x)$) and implicit flows (pc) are permitted to the level of the specified output channel (rule IO-TT-OUTPUT). For an

$$\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\}\,\Gamma[x \mapsto pc \sqcup t]} \qquad \text{(IO-TT-WRITE)}$$

$$\frac{pc \sqsubseteq \ell}{pc \vdash \Gamma\{input(x, \ell)\}\,\Gamma[x \mapsto \ell]} \qquad \text{(IO-TT-INPUT)}$$

$$\frac{pc \sqcup \Gamma(x) \sqsubseteq \ell}{pc \vdash \Gamma\{output(x, \ell)\}\,\Gamma} \qquad \text{(IO-TT-OUTPUT)}$$

Figure 15: Flow-sensitive typing rules with I/O (selected rules).

input command $input(x, \ell)$, the level of variable x is updated to ℓ if the program context does not make an illegal implicit flow (rule IO-TT-INPUT). Otherwise, it might violate soundness of the enforcement mechanism for programs like Figure 16, where the execution of an input command in a high context influences the received value of the next input command at the same level.

```
if High.h then input(Low.x,{L}) else skip;
input(Low.y,{L});
output(Low.y,{L});
```

Figure 16: An example that shows an implicit flow by input commands.

Running example Given the policy specified in the running example, the type system rejects the transformed program shown in Figure 14. The initial types of the variables are the singleton set of the nontransitive security label. Following the typing rules, the types of the variables `Alice.data_temp` and `Bob.data1_temp` are (at least) $\{A\}$. The rule for output commands demands that the specified level of the output value must be higher than union of the level of the program context and the level of variable x. The *if* branch is well-typed because $\{A\} \sqcup \{B\} \sqsubseteq \{A, B\}$, yet the type system cannot offer a suitable type for the *else* branch where $\{A\} \sqcup \{B\} \not\sqsubseteq \{B, C\}$.

Theorem 8 states soundness of the type system. If a transformed program is well-typed, then it satisfies the transitive noninterference, and by the result of Theorem 7, the original program complies with the corresponding nontransitive policy.

Theorem 8 (*Soundness of Flow-Sensitive Type System for Programs with I/O*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Transform(c)\}\,\Gamma' \;\Rightarrow\; TNI_{PI}(\mathcal{T}, Transform(c)).$$

V. Case study with JOANA

We develop a prototype of our transpiler to analyze Java programs. We follow the architecture illustrated in Figure 8 to implement a program canonicalizer and an input script generator for JOANA [11], a flow-sensitive information-flow analyzer for Java programs. The transpiler gets a path to a Java project and generates the canonical version of the program using Spoon [21], a library for transforming Java programs. The user defines a nontransitive policy by labeling the components (i.e., classes) of the program. Then, our tool generates a script as the input of JOANA, which detects possible illegal flows in the program. Our proof-of-concept implementation can support as many programs as JOANA may allow, as long as they are batch-job programs.

We evaluate our tool on four examples of nontransitive policies to demonstrate the benefits of the reduction from nontransitive to transitive policies in practice: Alice-Bob-Charlie (the running example), Confused deputy, Bank logger, and Low-High. The source code and materials of case studies are available online [1]. We discuss the details of transpilation and the JOANA's script for the running example, and to conserve space, we only report analysis results for the next cases. In Appendix B, the source code of the programs in question is presented.

A. Alice-Bob-Charlie (the running example)

We start with the running example as the first case, introduced in Figure 1. To model the batch-job style, we modify the code to include instances of components as fields of Java classes. Following standard practices in object-oriented programming, our prototype leverages *composition* relationship [24] between classes where an object is a part of another object. This leads to a hierarchy of objects, in which each object is responsible for creation and deletion of required objects of other classes. Assuming that no local variable creates a new instance of a class, the execution starts from the main object and continues in the underlying ones.

Given the main method as the starting point of the program, constructors naturally provide placeholders for inserting the init section (*initiator* methods), while the last line of the main method is the placeholder for final assignments existing in *finalizer* methods. By calling the finalizer method of the main object, following the composition hierarchy, objects invoke the finalizers as a chain. In the end, all of the *sink* fields are assigned.

The transpiler suffices to inject the initiator and finalizer methods per class. For readability, we slightly modify the canonicalization algorithm. We add a *source* field assigned to the initial value of the field in the original program, instead of replacing occurrences of the variable with *temp* variables. As an example, the canonical version of the program is shown in Figure 17.

Considering the labels A, B, and C, for `Alice`, `Bob`, and `Charlie` and with respect to the permitted flow $(A \trianglerighteq B, B \trianglerighteq C)$, the transpiler also generates the input script for JOANA. Figure 18 displays the important snippet of it.

The first line describes the power-lattice, where `e` denotes the empty set as the bottom element. It is followed by the list of annotations on field variables to distinguish *source*s and *sink*s of information per class. For example, the line `sink Charlie.data_sink BC` means `Charlie.data_sink` is a sink variable with the security level *BC* (the set of nontransitive labels can flow to C). The last command of the script triggers the flow-sensitive information flow analysis. As the result of the analysis, JOANA reports the security violation `Illegal flow from Alice.data_source to`

```java
public class Alice {
  private int data_source = 0,data,data_sink;
  private Bob b;
  public void initiator(){
    data = data_source;
  }
  public Alice(){
    initiator();
    b = new Bob();
  }
  public static void main(String[] args){
    Alice a = new Alice();
    a.operation();
    a.finalizer();
  }
  private void operation(){
    b.receive(data);
    b.good();
    b.bad();
  }
  public void finalizer(){
    data_sink = data;
    b.finalizer();
  }
}
```

```java
public class Bob {
  private int data1_source=0,data1,data1_sink;
  private int data2_source=1,data2,data2_sink;
  private Charlie c;
  public void initiator(){
    data1 = data1_source;
    data2 = data2_source;
  }
  public Bob(){
    initiator();
    c = new Charlie();
  }
  public void receive(int x){ data1 = x; }
  public void good(){ c.receive(data2); }
  public void bad(){ c.receive(data1); }
  public void finalizer(){
    data1_sink = data1;
    data2_sink = data2;
    c.finalizer();
  }
}
```

```java
public class Charlie {
  private int data_source, data, data_sink;
  public void initiator(){data = data_source;}
  public Charlie(){ initiator(); }
  public void receive(int x){ data = x; }
  public void finalizer(){ data_sink = data; }
}
```

Figure 17: The canonical version of Alice-Bob-Charlie.

`Charlie.data_sink, visible for BC`, which captures the undesired explicit flow.

Omitting invocation of the `bad` method yields a secure program. In this case, JOANA reports `No violations found` after running the same script on the canonical version of the secure program.

```
setLattice e<=A,e<=B,e<=C,A<=AB,A<=AC,B<=AB,
   B<=BC,AB<=ABC,C<=AC,C<=BC,AC<=ABC,BC<=ABC
source Alice.data_source   A
sink   Alice.data_sink     A
source Bob.data1_source    B
sink   Bob.data1_sink      AB
source Bob.data2_source    B
sink   Bob.data2_sink      AB
source Charlie.data_source C
sink   Charlie.data_sink   BC
run    classical-ni
```

Figure 18: A snippet of JOANA script for Alice-Bob-Charlie.

B. Confused deputy

We benefit from the fact that nontransitive information flow control supports enforcing both confidentiality and integrity policies. The confused deputy problem [12] occurs in a situation when an untrusted component is able to manipulate a trusted component and misuse its authority to execute a sensitive operation. It is an integrity problem since the policy states if the attacker is not permitted to alter a resource, then there must not be any way to do so, directly or by using a deputy. We adopt Lu and Zhang's code [17] as a starting point to represent the confused deputy problem.

Figure 19 illustrates the skeleton of the source code. We make use of four classes: `Library`, `Service`, `Downloaded_Code`, and `Trusted_Code`. Values in `Library` are protected and only `Service` is privileged to access them. The class `Downloaded_Code` is third-party code that cannot access to `Library`, while `Trusted_Code` is completely trusted. Invoking `addLog` method of `Service` is permitted because it updates a non-executable log file in `Service`, but the `process` method of `Library` must not be called with data from `Downloaded_Code` via `Service`. To rephrase the integrity policy, `Downloaded_Code` should not have any effects on the sensitive component `Library`, directly or indirectly, while `Trusted_Code` can. Given the initial letters of the component names as their labels, the specified policy is $D \trianglerighteq S$, $S \trianglerighteq L$, $T \trianglerighteq S$ and $T \trianglerighteq L$.

On the other hand, `Downloaded_Code` must not retrieve `Library`'s information through invoking the `query` method by `Service`. Taking confidentiality policies into account, we add flow relations $L \trianglerighteq S$, $S \trianglerighteq D$, $L \trianglerighteq T$, and $S \trianglerighteq T$ to exclude the illegal flows from `Library` to `Downloaded_Code` violating data secrecy. To sum up, the intended policy is the aggregation of the integrity and confidentiality policies, which are defined uniformly by the aforementioned nontransitive flows.

The transpiler generates the canonical version of the program and annotates sources and sinks of information in classes. JOANA discovers the violations in the program and reports the two existing illegal flows: `Illegal flow from Downloaded_Code.data_source to Library.printValue_sink, visible for LS` (integrity) and `Illegal flow from Library.someValue_source to Downloaded_Code.result_sink, visible for DS` (confidentiality).

```java
public class Library {
  private int someValue = 5, printValue = 0;
  ...
  public void process(int src){
    printValue = src;
  }
  public int retrieve(int key){
    return someValue;
  }
}
```

```java
public class Service {
  private int logFile = 0;
  private Library library;
  ...
  public void addLog(int x, int y){
    logFile += x + y ;
  }
  public void print(int data){
    library.process(data);
  }
  public int query(int key){
    return library.retrieve(key);
  }
}
```

```java
public class Downloaded_Code {
  private int data = 7, key = 4, result;
  private Service service;
  ...
  public static void main(String[] args){
   Downloaded_Code dc = new Downloaded_Code();
   dc.operation();
  }
  private void operation(){
   service.addLog(data, key);
   service.print(data);
   result = service.query(key);
  }
}
```

Figure 19: The skeleton of Confused deputy source code.

A secure version of the program is the one without calling `service.print(data)` and `service.query(key)` in the `operation` method. Now information from `Downloaded_Code` (as $\{D\}$) influences only `logFile` in `Service` (as {D, L, S, T}), which is allowed by the policy. JOANA also confirms security of the program by running the same script on the canonical version of the revised program.

C. Bank logger

We discuss another example in which two bank services for processing customers' information (`Bank`) and logging their public information (`Logger`) are totally separated. A client component (`BankLog`) is developed to communicate with both services at the same time. Figure 20 focuses on the important parts of the source code. The two components `Bank` and `BankLog` can mutually access each other's information, although `Logger` may read insensitive information. Thus, `Logger` must not interfere with `Bank` directly or indirectly. We label `Bank`, `Logger`, and `BankLog` components as B, L, and C, respectively. Consequently, the intended policy is $C \trianglerighteq B$, $B \trianglerighteq C$, and $C \trianglerighteq L$.

```java
public class Bank {
  private int id = 20;
  ...
  public int getBalance(int x){
    if (x == id) return balance; //flow #1
    return 0;
  }
}
```

```java
1   public class BankLog {
2     private int userId = 20, balance;
3     private Bank b; private Logger l;
4     ...
5     private void operation(){
6       balance = b.getBalance(userId);
7         if (balance > 0) //flow #2
8           l.append(userId);
9     }
10  }
```

Figure 20: An excerpt from Bank logger source code.

The current implementation of the program violates the policy by two implicit flows. The `getBalance` method checks whether the `id` exists, and `BankLog` only requests for logging if the sensitive value `balance` is positive. Executing the JOANA script on the canonical version of the program generates the following report: `Illegal flow from Bank.id_source to Logger.logFile_sink, visible for CL` (flow #1) and `Illegal flow from Bank.balance_source to Logger.logFile_sink, visible for CL` (flow #2).

To secure the program, the log content must not be influenced by sensitive information. One possible way to repair the program is logging the number of accesses to the client component `BankLog`. Hence, we replace lines 7 and 8 of `BankLog` (in the `operation` method) with `l.append(1)`. With this change, JOANA accepts the canonical version of the program using the same script.

D. Low-High

The previous examples included more than two components, which allowed us to contrast transitive and nontransitive policies. The following example demonstrates the compatibility with the baseline case of the two-level security policy. The program (in Appendix B) contains two components `Alice` and `Bob`, where `Alice` updates her data influenced by `Bob`'s secret value. We define the nontransitive policy $L \trianglerighteq H$ such that L is the label of `Alice` and H is for `Bob`.

```
setLattice e<=L,e<=H,L<=LH,H<=LH
source Alice.data_source L
sink   Alice.data_sink   L
source Bob.secret_source H
sink   Bob.secret_sink   LH
source Bob.data_source   H
sink   Bob.data_sink     LH
```

Figure 21: A snippet of JOANA script for Low-High.

The transpiler transforms the program and generates the input script for JOANA, as can be seen in Figure 21. Therefore, JOANA analyzes the program and reports message `Illegal flow from Bob.secret_source to Alice.data_sink, visible for L` expresses the security violation caused by the implicit flow.

Removing the illegal flow (line 13 in `Alice`) makes the program secure, which is verified by running the JOANA script on the canonical version of the modified program.

VI. Alternative policies and encodings

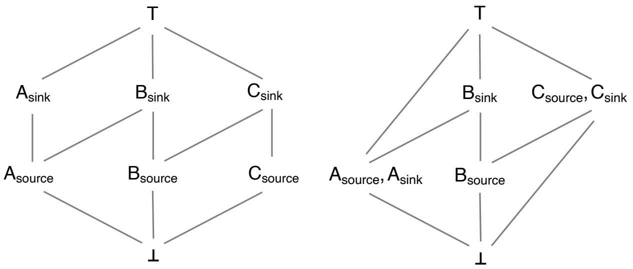
Fine-grained policies While the main motivation for nontransitive types is enforcing coarse-grained information-flow policies, where labels represent components, the notion of nontransitive security is not limited to module separation [17]. Other real-world scenarios such as policies in social media (e.g., "only my friends can see my photo but not friends of my friends") also naturally match nontransitive policies. Our framework can thus be generalized to decouple the flow-to relation from component labels, allowing fine-grained nontransitive policies.

Scalability The proposed transpiler employs the power-lattice encoding that expands the number of security levels exponentially. For the type system, however, its time and space complexity do not depend on the size of the lattice. The reason is that we never need to store the lattice, as the flow-to relation is implicitly derived from its elements. In an off-the-shelf deployment of JOANA, there is no time blowup, but we cannot avoid the space blowup because JOANA is lattice-agnostic. Making JOANA aware of the power-lattice nature of the lattice (e.g., in the style of DLM [19]) can help avoiding the blowup in the current implementation.

Alternative encodings A power-lattice encoding enables us to support declassification and dynamic policies. However, when such generality is not needed, we can reduce the size of the lattice by alternative encodings, with the cost of losing granularity of information stored in security labels.

We identify the soundness constraint for a nontransitive-to-transitive policy encoding as $\ell \trianglerighteq \ell' \iff \ell_{source} \sqsubseteq \ell'_{sink}$, where source and sink variables of a component are labeled as ℓ_{source} and ℓ_{sink}, respectively, when the component has label ℓ in the nontransitive setting (recall that $\trianglerighteq$ is reflexive). Note that the powerset lattice encoding indeed meets the condition because $\forall \ell, \ell' \in L_{\mathcal{N}}.\ell_{source} = \{\ell\} \wedge \ell_{sink} = C(\ell) \wedge \big(\ell \trianglerighteq \ell' \iff \{\ell\} \subseteq C(\ell')\big)$ (see Figure 6). Among various lattices satisfying the constraint, a minimal one is desirable, i.e., the one with the smallest set of labels.

We present a so-called *source-sink* lattice encoding that satisfies the soundness constraint and reduces the size of the lattice from exponential to polynomial. We start with a source-sink partial order where for all $\ell \in L_{\mathcal{N}}$, there are $\ell_{src}, \ell_{snk} \in L_{\mathcal{T}}$ such that $\ell_{src} \sqsubseteq \ell_{snk}$, due to reflexivity of the $\trianglerighteq$ relation. Then, according to the soundness constraint, we include transitive relations between levels based on the specified nontransitive flows. Since the security levels must constitute a lattice, we apply the Dedekind–MacNeille completion algorithm [4] to compute the smallest lattice containing the partial order. If a unique least upper (resp. greatest lower) bound for any pairs of source (resp. sink) levels does not exist, it adds an intermediary level between two source and two sink levels such that the intermediary level is the lub of the source levels and the glb of the sinks. It also makes one top and one bottom element for the lattice. Figure 22a illustrates the resulting source-sink lattice for the running example ($A \trianglerighteq B$ and $B \trianglerighteq C$).



(a) (b)

Figure 22: (a) A source-sink lattice encoding for the running example; (b) A minimal lattice.

In the worst case, the size of the lattice is $O(|L_{\mathcal{N}}|^2)$ and the time complexity of the algorithm is $O(|L_{\mathcal{N}}|^4)$, as proved in Appendix A. Furthermore, optimization techniques can make the partial order compact, before constructing the lattice out of it; for example, any pairs of ℓ_{src} and ℓ_{snk} coincide in the partial order when one of them is only in relation with the other one, not any other levels. Figure 22b depicts the minimal source-sink lattice for the nontransitive policy in question; observe how A_{sink} and C_{source} are collapsed.

We demonstrate the NTNI-to-TNI tranpilation defined for a source-sink lattice, in comparison with the power-lattice encoding, by replacing $\{\ell\}$ with ℓ_{src} and $C(\ell)$ with ℓ_{snk} in the labeling function and program transformation. In Appendix A, we formally introduce the transpilation using a source-sink lattice. We make use of the program canonicalization for batch-job programs and define the transitive encoding of a nontransitive policy based on a given source-sink lattice (Definition 15). We prove that any nontransitive policy on a program can be reduced to a corresponding transitive policy on a semantically equivalent program (Theorem 9). For the enforcement mechanism, we prove that the presented flow-sensitive type system, while a source-sink lattice is in place, is sound and more permissive than the nontransitive type system (Theorems 10 and 11). Moreover, our results can be generalized to programs with intermediate inputs and outputs, where the program transformation algorithm replaces the level of input and output commands to ℓ_{src} and ℓ_{snk}, respectively (Algorithm 3 and Theorem 12). We also prove that the flow-sensitive type system for programs with I/O is compatible with a source-sink lattice (Theorem 13).

VII. Related work

Our starting point is the special-purpose notions Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) by Lu and Zhang [17]. Our work demonstrates how to cast NTNI as classical noninterference on a lattice and how to improve the precision of NTT by classical flow-sensitive analysis.

Nontransitive noninterference is not to be confused by intransitive noninterference. Intransitive noninterference was introduced by Rushby [25] and explored by, amongst others, Roscoe and Goldsmith [23], Mantel and Sands [18], and Ron van der Meyden [30]. Intransitive noninterference is intended to address the *where* dimension of declassification [27]. The typical scenario for intransitive noninterference is ensuring that sensitive data is passed through a trusted encryption module before it is released. For example, security labels might be *low*, *encrypt*, and *high*, ordered by *high* → *encrypt* → *low* while *high* ↛ *low*. Like nontransitive policies, intransitive policies do not assume transitive policies. However, there is a fundamental difference between nontransitive and intransitive policies: intransitive noninterference allows *low* information to be (indirectly) dependent on *high*. In the encryption module scenario, this means that changes in the (high) plaintext may reflect in the changes in the (low) ciphertext. In contrast, nontransitive policy $A \trianglerighteq B$ and $B \trianglerighteq C$ guarantees that there are no information dependencies from A to C whatsoever.

Further approaches to declassification introduce decentralized hierarchies and dynamic policies. Myers and Liskov's DLM [19] is based on transitive policies that encode ownership in the labels. The goal is to allow declassification only if it is allowed by the owner(s) of the data. DC labels [28] by Stefan et al. models a setting of mutual distrust without relying on a centralized principal hierarchy. DC labels incorporate formulas over principals, modeling can-flow-to relation by logical implication. FLAM [2] by Arden et al. explores robust authorization to mitigate delegation loopholes in policies like DLM. Jia and Zdancewic [15] encode security types using authorization logic in a programming language for access control. Their encoding does not assume transitivity and it needs to be encoded as explicit delegations. Swamy et al. [29] and Broberg et al. [6] explore the effects of dynamic policy updates on the transitivity of flows. Broberg et al. call a flow *time-transitive* if information leaks from A to C via B even if no flows from A to C are allowed at any given time. This can happen when the policy of allowing flows from A to B is dynamically updated to allow flows from B to C. Time-transitivity is not in the scope of our work because our policies are static.

Rajani and Garg [22] explore the granularity of policies for information flow control. They show that fine-grained type systems that track the propagation of values are as expressive as coarse-grained type systems that track the propagation of context. Vassena et al. [31] expand the study to the dynamic setting. Xiang and Chong [33] use opaque labeled values in their study of dynamic coarse-grained information flow control for Java-like languages. However, in both cases, the considered policies are transitive. An interesting avenue for future work is to explore whether these approaches can be integrated with ours to be able to handle nontransitive policies.

Our proof-of-concept implementation of the flow-sensitive analysis for Java draws on Hammer and Snelting's JOANA [10], [11]. Note that our reduction results are general, enabling the use of other practical flow-sensitive analyses like Pidgin [16] by Johnson et al. for tracking nontransitive policies.

VIII. Conclusion

In order to support module-level coarse-grained information-flow policies, Nontransitive Noninterference (NTNI) and Nontransitive Types (NTT) have been suggested recently as a new security condition and enforcement. In contrast to Denning's classical lattice model, NTNI and NTT assume no transitivity of the underlying flow relation. NTNI and NTT, in the form they were proposed, are nonstandard, requiring the development of nonstandard semantic machinery to reason about NTNI and the development of nonstandard enforcement techniques to track NTT.

This paper demonstrates that despite the different aims and intuitions of nontransitive policies compared to classical transitive policies, nontransitive noninterference can in fact be reduced to classical transitive noninterference.

On the security characterization side, we show that NTNI corresponds to classical noninterference on a lattice that records source-to-sink relations derived from nontransitive policies. On the enforcement side, we devise a lightweight program transformation that enables us to leverage standard flow-sensitive information-flow analyses to enforce nontransitive policies. Further, we improve the permissiveness over the nonstandard NTT enforcement while retaining the soundness. We show that our security characterization and enforcement results naturally generalize to a language with intermediate input and outputs. An immediate practical benefit of our work is the implication that there is no need for dedicated design and implementation for the enforcement of nontransitive policies for practical programming languages. Instead, we can leverage state-of-the-art flow-sensitive information-flow tools, which we demonstrate by utilizing JOANA to enforce nontransitive policies for Java programs.

Acknowledgments Thanks are due to Yi Lu and Chenyi Zhang for inspiring this line of work and for the interesting discussions. This work was partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and the Danish Council for Independent Research for the Natural Sciences (DFF/FNU, project 6108-00363).

References

[1] M. M. Ahmadpanah, A. Askarov, and A. Sabelfeld. Nontransitive Policies Transpiled - Supplementary Materials. https://www.cse.chalmers.se/research/group/security/ntni, 2021.

[2] O. Arden, J. Liu, and A. C. Myers. Flow-limited authorization. In *CSF*, 2015.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, 2008.

[4] K. Bertet, M. Morvan, and L. Nourine. Lazy completion of a partial order to the smallest lattice. In *Second Int. Symp. on Knowledge Retrieval, Use and Storage for Efficiency*, 1997.

[5] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *CCS*, 2009.

[6] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *CSF*, 2015.

[7] S. Dahlgaard, M. B. T. Knudsen, and M. Stöckel. Finding even cycles faster via capped k-walks. In *STOC*, 2017.

[8] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[9] B. Ganter and S. O. Kuznetsov. Stepwise construction of the dedekind-macneille completion (research note). In *ICCS*, volume 1453, 1998.

[10] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 2009.

[11] C. Hammer and G. Snelting. JOANA: Java Object-sensitive ANAlysis. https://pp.ipd.kit.edu/projects/joana/, 2020.

[12] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 22(4), 1988.

[13] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*. 2012.

[14] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL*, 2006.

[15] L. Jia and S. Zdancewic. Encoding information flow in Aura. In *PLAS*, 2009.

[16] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *PLDI*, 2015.

[17] Y. Lu and C. Zhang. Nontransitive security types for coarse-grained information flow control. In *CSF*, 2020.

[18] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, 2004.

[19] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.

[20] L. Nourine and O. Raynaud. A fast incremental algorithm for building lattices. *J. Exp. Theor. Artif. Intell.*, 14(2-3), 2002.

[21] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. SPOON: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exp.*, 46(9), 2016.

[22] V. Rajani and D. Garg. Types for information flow control: Labeling granularity and semantic models. In *CSF*, 2018.

[23] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW*, 1999.

[24] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[25] J. Rushby. *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory Menlo Park, 1992.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[27] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comp. Sec.*, 2009.

[28] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, 2011.

[29] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW*, 2006.

[30] R. van der Meyden. What, indeed, is intransitive noninterference? *J. Comput. Secur.*, 2015.

[31] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan. From fine- to coarse-grained dynamic information flow control and back. In *POPL*, 2019.

[32] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comp. Sec.*, 1996.

[33] J. Xiang and S. Chong. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *S&P*, 2021.

[34] R. Yuster and U. Zwick. Finding even cycles even faster. *SIAM J. Discret. Math.*, 10(2), 1997.

Expression Evaluation

$$\frac{}{\langle v, M\rangle \Downarrow v} \qquad \text{(IO-VALUE)}$$

$$\frac{}{\langle x, M\rangle \Downarrow M(x)} \qquad \text{(IO-READ)}$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \qquad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2} \qquad \text{(IO-OPERATION)}$$

Command Evaluation

$$\frac{}{\langle skip, M, I, O\rangle \rightarrow \langle stop, M, I, O\rangle} \qquad \text{(IO-SKIP)}$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad M' = M[x \mapsto v]}{\langle x := e, M, I, O\rangle \rightarrow \langle stop, M', I, O\rangle} \qquad \text{(IO-WRITE)}$$

$$\frac{c = if\ e\ then\ c_{true}\ else\ c_{false} \qquad \langle e, M\rangle \Downarrow b}{\langle c, M, I, O\rangle \rightarrow \langle c_b, M, I, O\rangle} \qquad \text{(IO-IF)}$$

$$\frac{c = while\ e\ do\ c_{body} \qquad \langle e, M\rangle \Downarrow true}{\langle c, M, I, O\rangle \rightarrow \langle c_{body}; c, M, I, O\rangle} \qquad \text{(IO-WHILE-T)}$$

$$\frac{c = while\ e\ do\ c_{body} \qquad \langle e, M\rangle \Downarrow false}{\langle c, M, I, O\rangle \rightarrow \langle stop, M, I, O\rangle} \qquad \text{(IO-WHILE-F)}$$

$$\frac{\begin{array}{c} c = input(x, \ell) \qquad I(\ell) = v.\sigma \\ I' = I[\ell \mapsto \sigma] \qquad M' = M[x \mapsto v] \end{array}}{\langle c, M, I, O\rangle \rightarrow \langle stop, M', I', O\rangle} \qquad \text{(IO-INPUT)}$$

$$\frac{\begin{array}{c} c = output(x, \ell) \\ M(x) = v \qquad O' = O.v_{\ell} \end{array}}{\langle c, M, I, O\rangle \rightarrow \langle stop, M, I, O'\rangle} \qquad \text{(IO-OUTPUT)}$$

$$\frac{\langle c_1, M, I, O\rangle \rightarrow \langle c_1', M', I', O'\rangle}{\langle c_1; c_2, M, I, O\rangle \rightarrow \langle c_1'; c_2, M', I', O'\rangle} \qquad \text{(IO-SEQ-I)}$$

$$\frac{}{\langle stop; c, M, I, O\rangle \rightarrow \langle c, M, I, O\rangle} \qquad \text{(IO-SEQ-II)}$$

Figure 23: Language semantics with I/O.

APPENDIX

A. Source-sink encoding

We define the source-sink lattice encoding of a nontransitive policy to a transitive policy for canonical programs as follows.

Definition 15 (*Transitive Encoding of Nontransitive Policies*)**.** *Given a nontransitive policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$ and a program c, a corresponding transitive policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ on the canonical version of the program is $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}. \ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ ($\trianglerighteq$ is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq\rangle$ constitutes a lattice, and*

$$\forall x \in Var_c.\ \Gamma_{\mathcal{N}}(x) = \ell \implies \begin{cases} \Gamma_{\mathcal{T}}(x) & = & \ell_{src} \\ \Gamma_{\mathcal{T}}(x_{temp}) & = & \top \\ \Gamma_{\mathcal{T}}(x_{sink}) & = & \ell_{snk} \end{cases}.$$

As stated in Definition 15, the initial and final values of an ℓ-observable variable x of the given program are ℓ_{src}-

$$\frac{}{\Gamma \vdash v : \bot} \qquad \text{(IO-TT-VALUE)}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(IO-TT-READ)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \oplus e_2 : t_1 \sqcup t_2} \qquad \text{(IO-TT-OPERATION)}$$

$$\frac{}{pc \vdash \Gamma\{skip\}\,\Gamma} \qquad \text{(IO-TT-SKIP)}$$

$$\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\{x := e\}\,\Gamma[x \mapsto pc \sqcup t]} \qquad \text{(IO-TT-WRITE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \\ pc \sqcup t \vdash \Gamma\{c_{true}\}\,\Gamma' \\ pc \sqcup t \vdash \Gamma\{c_{false}\}\,\Gamma'\end{array}}{pc \vdash \Gamma\{if\ e\ then\ c_{true}\ else\ c_{false}\}\,\Gamma'} \qquad \text{(IO-TT-IF)}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma\{c_{body}\}\,\Gamma}{pc \vdash \Gamma\{while\ e\ do\ c_{body}\}\,\Gamma} \qquad \text{(IO-TT-WHILE)}$$

$$\frac{pc \vdash \Gamma\{c_1\}\,\Gamma' \qquad pc \vdash \Gamma'\{c_2\}\,\Gamma''}{pc \vdash \Gamma\{c_1;\, c_2\}\,\Gamma''} \qquad \text{(IO-TT-SEQ)}$$

$$\frac{pc \sqsubseteq \ell}{pc \vdash \Gamma\{input(x, \ell)\}\,\Gamma[x \mapsto \ell]} \qquad \text{(IO-TT-INPUT)}$$

$$\frac{pc \sqcup \Gamma(x) \sqsubseteq \ell}{pc \vdash \Gamma\{output(x, \ell)\}\,\Gamma} \qquad \text{(IO-TT-OUTPUT)}$$

$$\frac{\begin{array}{c}pc_1 \vdash \Gamma_1\{c\}\,\Gamma_1' \\ pc_2 \sqsubseteq pc_1 \qquad \Gamma_2 \sqsubseteq \Gamma_1 \qquad \Gamma_1' \sqsubseteq \Gamma_2'\end{array}}{pc_2 \vdash \Gamma_2\{c\}\,\Gamma_2'} \qquad \text{(IO-TT-SUB)}$$

Figure 24: Flow-sensitive typing rules with I/O.

and ℓ_{snk}-observable in the canonical version, respectively. Also, only the top-level observer can see final values of internal *temp* variables, thus makes them $\top$-observable. The next lemma demonstrates that for any canonical program satisfying a nontransitive policy, the program also complies with a corresponding transitive policy and vice versa.

Lemma 5 (*From $NTNI_{TI}$ to TNI_{TI} for Canonical Programs*)**.** Any canonical program $Canonical(c)$ is secure with respect to a nontransitive security policy $\mathcal{N}$ where $\forall x \in Var_c.\, \Gamma_{\mathcal{N}}(x_{temp}) = \Gamma_{\mathcal{N}}(x_{sink}) = \Gamma_{\mathcal{N}}(x)$ if and only if the canonical program is secure according to a corresponding transitive security policy $\mathcal{T}$. We write $\forall c.\forall \mathcal{N}.\exists \mathcal{T}.\ NTNI_{TI}(\mathcal{N}, Canonical(c)) \iff TNI_{TI}(\mathcal{T}, Canonical(c))$.

Therefore, we prove that any nontransitive policy on a given program can be modeled as a transitive policy on the canonical version of the program.

Theorem 9 (*From $NTNI_{TI}$ to TNI_{TI}*)**.** For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo canoni-caliztion) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$, as specified in Definition 15, such that $NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c')$. Formally,

$$\forall \mathcal{N}.\, \forall c.\, \exists \mathcal{T}.\, \exists c'.\, c \simeq_C c' \wedge NTNI_{TI}(\mathcal{N}, c) \iff TNI_{TI}(\mathcal{T}, c').$$

The next theorem states that the flow-sensitive type system is sound; in other words, if the type system accepts a canonical program, then the program satisfies the transitive noninterference, and consequently, the original program complies with the nontransitive policy.

Theorem 10 (*Soundness of Flow-Sensitive Transitive Type System*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\{Canonical(c)\}\,\Gamma' \Rightarrow TNI_{TI}(\mathcal{T}, Canonical(c)).$$

The next theorem shows if a program is secure under the nontransitive type system, the flow-sensitive type system accepts the canonical version of the program as well.

Theorem 11 (*Flow-Sensitive Type System Covers Nontransitive Type System*)**.**

$$\mathcal{P}, \Gamma_1, pc \vdash c : t \Rightarrow pc \vdash \Gamma_2\{Canonical(c)\}\,\Gamma_3,$$

where $\forall x \in Var_c.\, \Gamma_3(x_{temp}) \sqsubseteq \bigsqcup_{\ell \in \Gamma_1(x)} \ell_{src} \wedge \mathcal{P}(x) = \ell \Rightarrow \Gamma_2(x) = \Gamma_3(x) = \ell_{src} \wedge \Gamma_2(x_{temp}) = \top \wedge \Gamma_2(x_{sink}) = \Gamma_3(x_{sink}) = \ell_{snk}$.

We also introduce the transpilation for programs with intermediate input/outputs. Similar to the batch-job style, we establish a source-sink lattice out of nontransitive labels, i.e., $L_{\mathcal{T}} \supseteq \{\ell_{src}, \ell_{snk} | \ell \in L_{\mathcal{N}}\} \cup \{\top, \bot\}$ and $\forall \ell, \ell' \in L_{\mathcal{N}}.\ell \trianglerighteq \ell' \iff \ell_{src} \sqsubseteq \ell'_{snk}$ ($\trianglerighteq$ is reflexive) such that $\langle L_{\mathcal{T}}, \sqsubseteq\rangle$ is a lattice. In the program transformation algorithm, only the levels of input and output commands are modified because the notion of progress-insensitive noninterference only focuses on the relation between program inputs and outputs.

Program transformation As explained in Algorithm 3, we label sources and sinks of information at a security level $\ell \in L_{\mathcal{N}}$ as ℓ_{src} and ℓ_{snk}, respectively. More precisely, we replace $input(x, \ell)$ commands with $input(x, \ell_{src})$, and also $output(x, \ell)$ commands with $output(x, \ell_{snk})$ in the program.

Algorithm 3: Transformation algorithm for programs with I/O.

```
Input  : Program $c$
Output: Program $Transform(c)$
foreach $x \in Var_c$ do
    $c\,[input(x, \ell) \mapsto input(x, \ell_{src})]$
    $c\,[output(x, \ell) \mapsto output(x, \ell_{snk})]$
end
$Transform(c) := c$
return $Transform(c)$
```

Obviously, the transformed version of a given program preserves the meaning and termination behavior of the original program, yet it changes the channel of output values. The input and output values at the level ℓ can be found on the input channel with label ℓ_{src} and the output channel labeled as ℓ_{snk} in the canonical version of the given program. The next lemma shows the semantic relation between a given program and the transformed one.

Lemma 6 (*Semantic Equivalence Modulo Transformation*)**.** For any program c, the semantic equivalence $\simeq_T$ between the programs c and $Transform(c)$ holds where $c \simeq_T c' \overset{def}{=} \forall M.\forall I.\exists I'.\big(\forall \ell.I(\ell) = I'(\ell_{src})\big) \wedge \langle c, M, I, \varnothing\rangle \rightsquigarrow O \wedge \langle c', M, I', \varnothing\rangle \rightsquigarrow O' \wedge O' = O\,[v_\ell \mapsto v_{\ell_{snk}}]$.

Then, we prove a nontransitive policy on a given program (with intermediate inputs/outputs) can be reduced to a transitive policy on the transformed version of the program.

Theorem 12 (*From $NTNI_{PI}$ to TNI_{PI}*)**.** For any program c and any nontransitive security policy $\mathcal{N} = \langle L_{\mathcal{N}}, \trianglerighteq, \Gamma_{\mathcal{N}}\rangle$, there exist a semantically equivalent (modulo transformation) program c' and a transitive security policy $\mathcal{T} = \langle L_{\mathcal{T}}, \sqsubseteq, \Gamma_{\mathcal{T}}\rangle$ where $c' = Transform(c)$, $\langle L_{\mathcal{T}}, \sqsubseteq\rangle$ is a corresponding source-sink lattice and $\forall x \in Var_c.\,\ell = \Gamma_{\mathcal{N}}(x) \implies \Gamma_{\mathcal{T}}(x) = \ell_{src}$ such that $NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c')$. Formally,

$$\forall \mathcal{N}.\,\forall c.\,\exists \mathcal{T}.\,\exists c'.\,c \simeq_T c' \wedge NTNI_{PI}(\mathcal{N}, c) \iff TNI_{PI}(\mathcal{T}, c').$$

Theorem 13 (*Soundness of Flow-Sensitive Type System for Programs with I/O*)**.**

$$pc \vdash \Gamma_{\mathcal{T}}\,\{Transform(c)\}\,\Gamma' \implies TNI_{PI}(\mathcal{T}, Transform(c)).$$

Proof of complexity of source-sink lattice encoding. We know that source levels are incomparable in the source-sink partial order, the same for sink levels. Thus, if there is not a quadruple of levels, two sources and two sinks, such that source levels are in relation with both of the sinks, then adding a top and a bottom element yields the smallest lattice. To do so, we detect cycles of length four in the undirected graph of the partial order. In the worst case, it takes $\binom{|L_{\mathcal{N}}|}{2}.O(|L_{\mathcal{N}}|^2) = O(|L_{\mathcal{N}}|^4)$ for the graph that has $2.|L_{\mathcal{N}}|$ nodes; $O(|L_{\mathcal{N}}|^2)$ for finding each cycle [34], [7], and $\binom{|L_{\mathcal{N}}|}{2}$ cycles exist at most. For each cycle, we add one intermediary level to the partial order, as the unique least upper (resp. greatest lower) bound of the source (resp. sink) levels. Hence, in the worst case, the resulting lattice adds $\frac{|L_{\mathcal{N}}|^2}{2} + 2$ more levels to the partial order, thus $O(|L_{\mathcal{N}}|^2)$ is the size of the lattice. It is also proven that the Dedekind-MacNeille completion takes $O(r^2)$ where r is the number of elements in the lattice [4], [9], [20], thus $O(|L_{\mathcal{N}}|^4)$. □

B. Case studies

Alice-Bob-Charlie

```java
public class Alice {
  private int data = 13;
  private Bob b;
  public Alice(){
    b = new Bob();
  }
  public static void main(String[] args){
    Alice a = new Alice();
    a.operation();
  }
  private void operation(){
    b.receive(data);
    b.good();
    b.bad();
  }
}
```

```java
public class Bob {
  private int data1 = 0, data2 = 42;
  private Charlie c;
  public Bob(){
    c = new Charlie();
  }
  public void receive(int x){
    data1 = x;
  }
  public void good(){
    c.receive(data2);
  }
  public void bad(){
    c.receive(data1);
  }
}
```

```java
public class Charlie {
  private int data;
  public Charlie(){ }
  public void receive(int x){
    data = x;
  }
}
```

Confused deputy

```java
public class Library {
  private int someValue = 5;
  private int printValue = 0;
  public Library(){ }
  public void process(int src){
    printValue = src;
  }
  public int retrieve(int key){
    return someValue;
  }
}
```

```java
public class Service {
  private int logFile = 0;
  private Library library;
  public Service(){
    library = new Library();
  }
  public void addLog(int x, int y){
    logFile += x + y ;
  }
  public void print(int data){
    library.process(data);
  }
  public int query(int key){
    return library.retrieve(key);
  }
}
```

```java
public class Downloaded_Code {
  private int data = 7, key = 4, result;
  private Service service;
  public Downloaded_Code(){
    service = new Service();
  }
  public static void main(String[] args){
    Downloaded_Code dc = new Downloaded_Code();
    dc.operation();
  }
  private void operation(){
    service.addLog(data, key);
    service.print(data);
    result = service.query(key);
  }
}
```

Bank logger

```
public class Bank {
  private int id = 20, balance = 100;
  public Bank(){ }
  public int getBalance(int x){
    if (x == id)
      return balance;
    return 0;
  }
}
```

```
public class Logger {
  private static int logFile;
  public Logger(){ }
  public void append(int x){
    logFile += x;
  }
}
```

```
public class BankLog {
  private int userId = 20, balance;
  private Bank b;
  private Logger l;
  public BankLog(){
    b = new Bank();
    l = new Logger();
  }
  public static void main(String[] args){
    BankLog bl = new BankLog();
    bl.operation();
  }
  private void operation(){
    balance = b.getBalance(userId);
    if (balance > 0)
      l.append(userId);
  }
}
```

Low-High

```
public class Bob {
  private int secret = 100, data;
  public Bob(){ }
  public void receive(int x){
    data = x;
  }
  public int getSecret(){
    return secret;
  }
}
```

```
public class Alice {
  private int data = 10;
  private Bob bob;
  public Alice(){
    bob = new Bob();
  }
  public static void main(String[] args){
    Alice a = new Alice();
    a.sendDataToBob();
  }
  public void sendDataToBob(){
    bob.receive(data);
    if (bob.getSecret() > data)
      data++;
  }
}
```