Impossibility of Precise and Sound Termination Sensitive Security Enforcements

Anonymous Author(s)

***Abstract*—An information flow policy is *termination sensitive* if it imposes that the termination behavior of programs is not influenced by confidential input. Termination sensitivity can be statically or dynamically enforced. On one hand, existing static enforcement mechanisms for termination sensitive policies are typically quite conservative and impose strong constraints on programs like absence of while loops whose guard depends on confidential information. On the other hand, dynamic mechanisms can enforce termination sensitive policies in a less conservative way. Secure Multi-Execution (SME) [1], one of such mechanisms, was even claimed to be *sound and precise* in the sense that the enforcement mechanism will not modify the observable behavior of programs that comply with the termination sensitive policy. However, termination sensitivity is a subtle policy, that has been formalized in different ways. A key aspect is whether the policy talks about *actual* termination, or *observable* termination.**

This paper proves that termination sensitive policies that talk about actual termination are *not enforceable in a sound and precise way*. For static enforcements, the result follows directly from a reduction of the decidability of the problem to the halting problem. However, for dynamic mechanisms the insight is more involved and requires a diagonalization argument.

In particular, our result contradicts the claim made about SME. We correct these claims by showing that SME enforces a subtly different policy that we call *indirect* termination sensitive noninterference and that talks about observable termination instead of actual termination. We construct a variant of SME that is sound and precise for indirect termination sensitive noninterference. Finally, we also show that static methods can be adapted to enforce indirect termination sensitive information flow policies (but obviously not precisely) by constructing a sound type system for an indirect termination sensitive policy.

I. Introduction

Information flow policies are widely studied security policies [2]. Roughly, they state that public outputs of a program are not influenced by confidential inputs. Dually, for integrity, trusted outputs are not influenced by untrusted inputs. Information flow policies restrict how information can flow in a program: information should not flow from *high* (confidential, respectively untrusted) inputs to *low* (public, respectively trusted) outputs. This can be generalized from high and low (a 2-element lattice) to an arbitrary lattice of security levels.

Many information flow policies exist. Different definitions cater for different programming models (e.g. sequential versus concurrent programming, or interactive versus batch-style programming), differ in the classes of information channels they consider (e.g. whether timing and/or termination channels are taken into account), or allow for intentional release of information (e.g. declassification or endorsement).

This paper focuses on the *termination sensitivity* aspect of information flow policies [3] in deterministic programs. Roughly, an information flow policy is termination sensitive, if (next to outputs also) termination of the program is not influenced by high inputs. Let us consider a simple termination-sensitive information flow policy called termination sensitive noninterference (TSNI) for sequential deterministic batch-style programs. Sequential deterministic batch-style programs take a tuple consisting of a low and a high input (i_L, i_H), and produce a tuple with a low and a high output (o_L, o_H). TSNI for such programs p can be formalized as: for all i_L, i_H, i'_H, the executions of p on (i_L, i_H) and on (i_L, i'_H) either (1) both terminate with the same low output, or (2) both diverge. In other words, the low output and the termination behavior of the program are independent of the high inputs.

Surprisingly, *termination sensitive information flow policies can not be enforced soundly and precisely*, at least in the case of deterministic programs. This is surprising because of previous results [1], [4]–[6] on TSNI mechanisms. The main contribution of this paper is a proof of impossibility for the deterministic case: for static termination sensitive information flow enforcement mechanisms, the result follows from a reduction of the decidability of the problem to the halting problem [7]. However, for dynamic mechanisms the insight is more involved and requires a diagonalization argument.

Another contribution of this paper is that we discovered that a dynamic enforcement mechanism called Secure Multi-Execution (SME) [1] claimed to enforce TSNI actually enforces a subtly different policy that we call Indirect TSNI (ITSNI). ITSNI says that the non-termination of a program due to specific high inputs can not be *observed* by a low observer, under some model of what a low observer can observe (for instance, such an observer can only observe the low outputs, and hence learns about termination indirectly in the sense that no more outputs are produced).

For the confidentiality interpretation of policies, the choice between TSNI and ITSNI is essentially a choice of attacker model: what exactly can an attacker observe about non-termination? If we assume attackers can observe actual computation, for instance because they can monitor the CPU load, then TSNI is the appropriate policy. But in many cases attackers learn about termination only indirectly, for instance by means of an explicit end-of-program output, or even more indirectly because no further output is produced. The results of this paper show that the difference between these attacker models is significant: while TSNI can not be enforced soundly and precisely, ITSNI *can*. The essential insight is that if the attacker learns about termination indirectly, then an enforcement mechanism can *fake* termination. For instance,

the enforcement mechanism can make attackers believe the program terminated by sending an end-of-program output to low observers, even while it is still doing work to compute high outputs. This power to fake termination is sufficient to construct sound and precise enforcement mechanisms.

For the integrity interpretation of policies, the choice between TSNI and ITSNI is essentially a choice of security objective. TSNI enforces that an attacker can not influence program termination by means of low (untrusted) inputs. This is a sensible security objective if one cares about the consumption of CPU cycles: malicious untrusted inputs should for instance not unexpectedly lead the program into an infinite loop. ITSNI enforces that untrusted inputs can not influence whether the program *appears* to be terminating with respect to trusted outputs, for instance malicious untrusted inputs can not unexpectedly lead to an infinite number of trusted outputs. This is a sensible security objective if one cares for instance about consumption of storage space for these trusted outputs.

For the remainder of the paper, we will mainly take the confidentiality perspective of policies, but all our results can be dualized for integrity.

In summary, the main contributions of this paper are:

- We prove an impossibility result: termination sensitive information flow policies can not be enforced soundly and precisely for deterministic programs.
- We correct some inaccuracies in the literature: we show that SME [1] previously claimed to soundly and precisely enforce TSNI actually does not. Even when the lattice is totally ordered, SME [1], in fact, enforces a weakened notion that we call Indirect TSNI, or ITSNI. Unfortunately, SME as presented in [1] does not *precisely* enforce ITSNI.
- We show the usefulness of ITSNI by constructing a dynamic enforcement mechanism (based on SME) which is sound and precise, as well as a static enforcement mechanism which is sound.

The remainder of this paper is structured as follows: Section II presents the impossibility result in a simple and general setting. Section III presents our language setting. Section IV presents the security policies for our language. Section V presents the definitions of enforcement mechanism in our language and instantiates the proof of Section II to the information flow policy of termination sensitive non interference, our language, and enforcement mechanism. Section VI explains the failure of SME with low-prio scheduler on enforcing soundly and precisely TSNI, and presents security guarantees of SME with low-prio scheduler. It also presents a sound and precise enforcement mechanism for ITSNI. Section VII presents a static enforcement mechanism for ITSNI. Section VIII discusses related work. We conclude in Section IX.

II. Impossibility Result

Our objective in this section is to prove that TSNI can not be enforced soundly and precisely. We prove this impossibility result for a very simple case: batch programs from natural numbers to natural numbers, where both the input number as well as the output number are labeled H. Hence TSNI for this case just says that the input should not influence termination behaviour, or in other words, such a program is TSNI if it either (1) diverges on all inputs or (2) converges on all inputs.

The fact that we can prove the impossibility result for such a simple case makes the result *stronger*. The proof for this simple case implies that it is also impossible to soundly and precisely enforce TSNI for more complex cases. For instance, if we were to have a sound and precise enforcement mechanism for batch programs that take both L and H inputs and produce L and H outputs, then, by applying that mechanism to the subset of programs that only take H inputs and produce H outputs, we would have again a sound and precise enforcement mechanism for the simple case – and we have proven that to be impossible.

A. Programs, policies and enforcement mechanisms

Without loss of generality, we consider a complete programming language for writing deterministic programs that compute functions from $\mathbf{N}$ to $\mathbf{N}$, where $\mathbf{N}$ is the set of the natural numbers.

We construct an enumeration of all programs in this language, and we write P_x to refer to the program at the x-th position in the enumeration. Each program P_x corresponds to a partial recursive function $\varphi_x : \mathbf{N} \to \mathbf{N}$, such that:

$$\varphi_x(y) = \begin{cases} z & \text{if } P_x(y) = z, \\ \text{undefined} & \text{if } P_x(y) \text{ divergent.} \end{cases}$$

Two partial functions φ_x and $\varphi_{x'}$ are equal (denoted by $\varphi_x = \varphi_{x'}$) if for all y, either $\varphi_x(y)$ and $\varphi_{x'}(y)$ are defined and equal or $\varphi_x(y)$ and $\varphi_{x'}(y)$ are not defined. We denote the range of φ_x as $range(\varphi_x)$.

We define a *policy* $\mathcal{P}$ to be a set of programs, to be thought of as the set of programs allowed by the policy. We require that if $P_x \in \mathcal{P}$ and P_x and $P_{x'}$ are equivalent (that is $\varphi_x = \varphi_{x'}$), then $P_{x'} \in \mathcal{P}$. We write $\varphi_x \in \mathcal{P}$ when $P_x \in \mathcal{P}$.

An *enforcement mechanism* is a mechanism to make sure that programs comply with a given policy. Such mechanisms can be static (like a type system), or dynamic (like a reference monitor or a taint tracker). By defining an enforcement mechanism to be a total recursive function from programs to programs, as in e.g. [8], [9], we cover all these cases: a static enforcement mechanism is a function that rejects some input programs, and returns the programs that pass the static check unmodified, whereas a dynamic enforcement mechanism is a function that inlines the necessary dynamic checks in the input program. By using our enumeration of programs, we can think of an enforcement mechanism as a function from $\mathbf{N}$ to $\mathbf{N}$.

Definition II.1. *An enforcement mechanism EM of $\mathcal{P}$ is a* total, recursive *function from* $\mathbf{N}$ *to* $\mathbf{N}$. *An enforcement mechanism is said to be* sound, *respectively* precise, *if it satisfies:*

- *Soundness:* $\forall x \in \mathbf{N} : \varphi_{EM(x)} \in \mathcal{P}$.
- *Precision:* $\forall x \in \mathbf{N} : \varphi_x \in \mathcal{P} \implies \varphi_{EM(x)} = \varphi_x$.

The soundness requirement makes sure that the mechanism actually enforces the policy: every program that is an output of the enforcement mechanism complies with the policy. Soundness is a very common requirement on both static and dynamic enforcement mechanisms.

The precision requirement constrains what the enforcement mechanism can do to programs that already satisfy the policy. In the ideal case, an enforcement mechanism would leave such programs untouched: the mechanism does not have to do anything if the program on which it is operating already satisfies the policy. However, since it is in general undecidable whether the input program satisfies the policy, an enforcement mechanism might insert some defensive checks anyway. Hence, it makes sense to only require that the enforcement mechanism transforms the program to some possibly other program, but one that is *indistinguishable* from the input program. While the program has changed syntactically by the insertion of the defensive checks, for a program that complies with the policy these defensive checks will not change the semantics of the program. The notion of indistinguishability that is used can depend on the kind of programs one is considering, but in our simple setting here, the only notion of indistinguishability that makes sense is extensional function equality, and that is what the definition above captures.

Static enforcement mechanisms are usually not precise, as most interesting policies are undecidable and hence a static mechanism will conservatively reject some programs that actually satisfy the policy. However, precise (and sound) dynamic mechanisms exist for interesting policies like noninterference [9].

If *EM* is a sound and precise enforcement mechanism for $\mathcal{P}$, we have that

$$EM(x) = \begin{cases} x' & \text{if } \varphi_x \in \mathcal{P}, \text{ where } \varphi_x = \varphi_{x'}, \\ x'' & \text{if } \varphi_x \notin \mathcal{P}, \text{ where } \varphi_{x''} \in \mathcal{P}. \end{cases}$$

B. Termination sensitivity can not be soundly and precisely enforced

We now formalize the very simple case of a TSNI policy for batch programs from $\mathbf{N}$ to $\mathbf{N}$ where both the input number and the output number are H, and we call it the *termination sensitive* policy.

Definition II.2 (Termination Sensitive Policy)**.** *A program satisfies the termination sensitive (TS) policy (denoted by $P \in TS$) if for all x_1 and x_2:*

$$(\exists y_1 \in \mathbf{N} : P(x_1) = y_1) \implies (\exists y_2 \in \mathbf{N} : P(x_2) = y_2)$$

It is straightforward to see from Definition II.2 that $P_x \in TS$ if and only if P_x either converges on all inputs or diverges on all inputs.

Now we are ready to prove our impossibility result.

Theorem II.1. *There is no sound and precise enforcement mechanism for TS.*

Proof. The proof is a relatively straightforward diagonalization argument similar to the proof that the set of all indices of total programs is not decidable [7].

Suppose that there is a sound and precise enforcement mechanism for TS. We have that:

$$EM(x) = \begin{cases} x' & \text{if } \varphi_x \in TS, \text{ where } \varphi_x = \varphi_{x'}, \\ x'' & \text{if } \varphi_x \notin TS, \text{ where } \varphi_{x''} \in TS. \end{cases}$$

Given such an *EM*, we can construct a function ψ whose range will be a set of indices of programs that are all total. We construct ψ as below:

$$\psi(x) = \begin{cases} EM(x) & \text{if } P_{EM(x)}(x) \text{ convergent}, \\ \text{divergent} & \text{if } P_{EM(x)}(x) \text{ divergent}. \end{cases}$$

Since *EM* is total and recursive, from Church's Thesis, it follows that ψ is a partial recursive function. Let $A = range(\psi)$ be the range of ψ, and $AConv \triangleq \{x | \forall y : \varphi_x(y) \text{ defined}\}$ be the set of all indices of all programs that converge on all inputs.

From the property of *EM* and the construction of ψ, we have that:

1) $x \in A \implies x \in AConv$,
2) $x \in AConv \implies \exists x' \in A : \varphi_x = \varphi_{x'}$,
3) A is infinite (notice that the set of all total recursive functions is infinite).

So we see that A is very close to the set of indices of all total programs. We can now use a diagonalization argument.

Since A is the range of a partial recursive function, from Corollary 5.V(a) in [7], A is recursively enumerable and hence, there is a total and recursive function f s.t. $A = range(f)$.

From f, we first construct a one-to-one function h with the same range as f, as defined below, where for an x, $\theta y[\forall i \leq x : f(y) \neq h(i)])$ returns the smallest value of y s.t. $\forall i \leq x : f(y) \neq h(i)$.

$$\begin{aligned} h(0) &= f(0) \\ h(x+1) &= f(\theta y[\forall i \leq x : f(y) \neq h(i)]) \end{aligned}$$

From (3), A is infinite. Therefore, the range of f is also infinite. Thus, h is total. Since f is total and recursive, and the construction process is recursive, from Church's Thesis, h is also recursive.

We show now that $range(h) = range(f) = A$. Indeed, from the construction, we have that $range(h) \subseteq range(f)$. We now need to prove that $range(f) \subseteq range(h)$, that is $\forall x : \exists y : f(x) = h(y)$.

- $x = 0$: the proof is trivial (since $f(0) = h(0)$).
- Assume that the statement holds for all $i \leq x$. That is $\forall i \leq x, \exists y : h(y) = f(i)$. We consider $x + 1$.
  - $f(x+1) \in \{f(0), \ldots, f(x)\}$. From IH, the statement holds.
  - $f(x+1) \notin \{f(0), \ldots, f(x)$. Suppose that $h(y_0) = f(x)$. We consider $h(y_0 + 1)$. Since $f(x+1) \notin \{f(0), \ldots, f(x)\}$, from IH, it follows that $f(x+1) \neq h(i)$ for all $i \leq y_0$. From the definition of h,

$h(y_0 + 1) = f(x + 1)$. The statement holds for this case.

Finally, we construct the function g that will help us establish a contradiction:

$$g = \lambda x.\varphi_{h(x)}(x) + 1$$

As proven above, $range(h) = A$. From (1), it follows that $h(x) \in ACONV$ and hence, $\varphi_{h(x)}$ is total and recursive. Since $\varphi_{h(x)}$ is total and recursive, from Church's Thesis, g is also total and recursive.

Let z_0 be the index of g. It follows that $z_0 \in ACONV$. From (2), there exists $z_1 \in A$ s.t. $\varphi_{z_0} = \varphi_{z_1}$. Since $z_1 \in A$ and h is one-one, $y_1 = h^{-1}(z_1)$ is defined. From the definition of g, we have that $g(y_1) = \varphi_{h(y_1)}(y_1) + 1$. From the definition of y_1, we also have that $g(y_1) = \varphi_{z_0}(y_1) = \varphi_{z_1}(y_1) = \varphi_{h(h^{-1}(z_1))}(y_1) = \varphi_{h(y_1)}(y_1)$. Since g is total, we have a contradiction. □

III. Language

The definition of program below is based on the description of programs presented in [1], where a program can consume input events from channels and send output events to channels.

Definition III.1 (Program)**.** *A program is a labeled transition system*

$$\langle \mathit{State}, \mathit{TerminatedState}, \mathit{InitialState}, \mathit{InputEvent}, \mathit{OutputEvent}, \mathit{Channel}, \twoheadrightarrow \rangle,$$

where State is the set of states, InitialState ⊆ State is the set of initial states, TerminatedState ⊆ State is the set of terminated states, Channel is the set of channels, InputEvent is the set of input events, OutputEvent is the set of output events, and the set of events is Act = InputEvent ∪ OutputEvent, and ↠ ⊆ State × Act × State is the transition relation.

For an input event $i \in InputEvent$, we write $i \in InputEvent_{ch}$ when i is an input event from channel ch. For an output event $o \in OutputEvent$, we write $o \in OutputEvent_{ch}$ when o is an output event from channel ch. The output event $\circledast \in OutputEvent$ is a special silent event and we assume it does not occur on any channel. An input event must originate from an input channel. An output event must either end on an output channel or be $\circledast$. We write $Q \xrightarrow{a} Q'$ when $\langle Q, a, Q' \rangle \in \twoheadrightarrow$.

We assume the following constraints on the transition relation:

- *for all $Q \in TerminatedState$, there is no a and Q' s.t. $Q \xrightarrow{a} Q'$.*
- *for all $Q \notin TerminatedState$,*
  - *there exist $a \in Act$ and Q' s.t. $Q \xrightarrow{a} Q'$,*
  - *for all $i \in InputEvent_{ch}$ and $a \in Act$, if $Q \xrightarrow{i} Q'$ and $Q \xrightarrow{a} Q''$ then $a \in InputEvent_{ch}$,*
  - *for all $i \in InputEvent_{ch}$, if $Q \xrightarrow{i} Q'$ and $Q \xrightarrow{i} Q''$ then $Q' = Q''$,*
  - *for all $o \in OutputEvent_{ch}$ and $a \in Act$, if $Q \xrightarrow{o} Q'$ and $Q \xrightarrow{a} Q''$ then $a = o$ and $Q' = Q''$.*

There are a few observations about programs. First, the program can always make progress unless it is terminated. Second, for each non-terminated state of a program, it can only consume an input event or generate an output event. We assume programs only have input-nondeterminism. In addition, we assume that if $Q \xrightarrow{a} Q'$, then Q generates event a and transitions to Q' in finite time.

We define a stream as the coinductive interpretation of the grammar $S ::= [] \mid s : S$, where $[]$ is the empty stream, and s ranges over stream elements. We write *fin*(S) when S is a finite stream, and *inf*(S) when S is an infinite stream. Two streams S and S' are equal (denoted by $S = S'$) if they are pairwise equal. The equality of streams is defined coinductively as below.

$$\frac{}{[] = []} \qquad \frac{s = s' \qquad S = S'}{s : S = s' : S'}$$

We use I and O as meta-variables for inputs and outputs of programs defined as functions from channels to respectively input event streams and output event streams. We write $I = I'$ when $dom(I) = dom(I')$ and for all *ch*, $I(ch) = I'(ch)$, where $dom(I)$ returns the domain of I. Similarly, we have $O = O'$.

The behavior of a state of a program can be considered as a function that transforms inputs to consumed inputs and generated outputs. Therefore, we define $Q(I) \rhd \langle I', O \rangle$ coinductively as below, where *terminated*(Q) means that Q is a terminated state, and for any channel *ch*, $I_{[]}(ch) = []$ and $O_{[]}(ch) = []$.

$$\frac{\mathit{terminated}(Q)}{Q(I) \rhd \langle I_{[]}, O_{[]} \rangle} \qquad \frac{Q \xrightarrow{\circledast} Q' \qquad Q'(I) \rhd \langle I', O \rangle}{Q(I) \rhd \langle I', O \rangle}$$

$$\frac{Q \xrightarrow{i} Q' \qquad i \in \mathit{InputEvent}_{ch} \qquad Q'(I) \rhd \langle I', O \rangle}{Q(I[ch \mapsto i : I(ch)]) \rhd \langle I'[ch \mapsto i : I'(ch)], O \rangle}$$

$$\frac{Q \xrightarrow{o} Q' \qquad o \in \mathit{OutputEvent}_{ch} \qquad Q'(I) \rhd \langle I', O \rangle}{Q(I) \rhd \langle I', O[ch \mapsto o : O(ch)] \rangle}$$

Lemma III.1. *For any Q and I, if $Q(I) \rhd \langle I', O \rangle$ and $Q(I) \rhd \langle I'', O' \rangle$, then $I' = I''$ and $O = O'$.*

Proof. By coinduction on the definition of $Q(I) \rhd \langle I', O \rangle$. □

A language for writing programs: We consider a language for writing programs with the syntax presented in [1]. Programs written in the presented language are instances of programs defined in Definition III.1. Commands in the language are those of the while language with the addition of input and output commands: a command to ask for an input event from a channel (i.e. **input** x **from** ch) and a command to send an output event to a channel (i.e. **output** e **to** ch). An expression is a value v, a variable x, or of the form $e \oplus e$, where $\oplus$ is a binary operator. Values v are integer numbers. An input event or an output event on channel *ch* is $ch(v)$ for some v.

(programs) $\quad P ::= \textbf{skip} \mid x := e \mid \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2$

$$\textsc{Assign}\ \frac{v = \mu(e)}{\langle x := e, \mu\rangle \xrightarrow{\circledast} \langle \textbf{skip}, \mu[x \mapsto v]\rangle}$$

$$\textsc{If1}\ \frac{\mu(e) = v \qquad v = 1}{\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu\rangle \xrightarrow{\circledast} \langle P_1, \mu\rangle}$$

$$\textsc{If2}\ \frac{\mu(e) = v \qquad v \neq 1}{\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu\rangle \xrightarrow{\circledast} \langle P_2, \mu\rangle}$$

$$\textsc{While1}\ \frac{\mu(e) = v \qquad v = 1}{\langle \textbf{while } e \textbf{ do } P, \mu\rangle \xrightarrow{\circledast} \langle P; \textbf{while } e \textbf{ do } P, \mu\rangle}$$

$$\textsc{While2}\ \frac{\mu(e) = v \qquad v \neq 1}{\langle \textbf{while } e \textbf{ do } P, \mu\rangle \xrightarrow{\circledast} \langle \textbf{skip}, \mu\rangle}$$

$$\textsc{Seq1}\ \frac{\langle P_1, \mu\rangle \xrightarrow{a} \langle P_1', \mu'\rangle}{\langle P_1; P_2, \mu\rangle \xrightarrow{a} \langle P_1'; P_2, \mu'\rangle} \qquad \textsc{Seq2}\ \frac{}{\langle \textbf{skip}; P_2, \mu\rangle \xrightarrow{\circledast} \langle P_2, \mu\rangle}$$

$$\textsc{In}\ \frac{}{\langle \textbf{input } x \textbf{ from } ch, \mu\rangle \xrightarrow{ch(v)} \langle \textbf{skip}, \mu[x \mapsto v]\rangle}$$

$$\textsc{Out}\ \frac{\mu(e) = v}{\langle \textbf{output } e \textbf{ to } ch, \mu\rangle \xrightarrow{ch(v)} \langle \textbf{skip}, \mu\rangle}$$

Fig. 1: Semantics of reactive programs

$$\begin{array}{lll} & & \mid \textbf{while } e \textbf{ do } P \mid c_1; c_2 \\ & & \mid \textbf{input } x \textbf{ from } ch \mid \textbf{output } e \textbf{ to } ch \\ \text{(expressions)} & e ::= & v \mid x \mid e \oplus e \\ \text{(input events)} & i ::= & ch(v) \\ \text{(output events)} & o ::= & ch(v) \mid \circledast \\ \text{(channels)} & & ch \end{array}$$

A state of a program P is of the form $\langle P, \mu\rangle$ where μ is a memory, a total function from variables in a set $\mathbf{Var}$ to values. An initial state of a program P is $\langle P, \mu_0\rangle$, where μ_0 is the initial memory. A state of the form $\langle \textbf{skip}, \mu\rangle$ is a terminated state. The semantics rules of programs are presented in Fig. 1, where $\mu(e)$ returns the result of the evaluation of e with μ. We assume that for all μ and e, there exists a unique v s.t. $\mu(e) = v$.

IV. Policies

As in [10], a policy is defined on a *state*, not on the whole program. Notice that a program starts executing from some initial state. Thus, a program satisfies a policy when all of its initial states satisfy the policy, and this should be the interpretation when we write a program satisfies a policy in examples in this section.

Next we define different versions of noninterference on a finite lattice $\langle \mathcal{L}, \sqsubset\rangle$, where $\mathcal{L}$ is a finite set of levels and $\sqsubset$ is the order between levels. For two levels l and l', we write $l \sqsubseteq l'$ when $l \sqsubset l'$ or $l = l'$. We write $\sqcup$ and $\sqcap$ for the least upper bound and greatest lower bound respectively. We represent by $\top$ the top level of the lattice.

Let Γ be a security environment, a total mapping from channels to levels. Given an input I, $I|_l$ returns the input with input event streams of channels at levels smaller than or equal to l. That is $\mathit{dom}(I|_l) = \{ch \in \mathit{dom}(I) | \Gamma(ch) \sqsubseteq l\}$, and for any channel $ch \in \mathit{dom}(I|_l)$, $I|_l(ch) = I(ch)$. Similarly, we have $O|_l$.

A. Termination-Sensitive Noninterference

TSNI [1] assumes that only when executions terminate, an observer at l can observe which input events on channels visible to him are consumed and which output events on channels visible to him are generated. Furthermore, an observer at l can observe the termination of executions. That is he can determine whether an execution is still in progress or not (even though there are no non-silent outputs).

Since $Q(I) \rhd \langle I', O\rangle$ does not give us information about whether the execution of Q on I terminates or not, we overload $\rhd$ and define $Q(I) \rhd A$ which captures all consumed input events and all generated output events (not just non-silent output events) by the execution of Q with I. $Q(I) \rhd A$ is defined coinductively as below, where A is a stream of input events and output events.

$$\frac{\mathit{terminated}(Q)}{Q(I) \rhd []} \qquad \frac{Q \xrightarrow{o} Q' \qquad Q'(I) \rhd A}{Q(I) \rhd o : A}$$

$$\frac{Q \xrightarrow{i} Q' \qquad i \in \mathit{InputEvent}_{ch} \qquad Q'(I) \rhd A}{Q(I[ch \mapsto i : I(ch)]) \rhd i : A}$$

Lemma IV.1. *For any Q, I, if $Q(I) \rhd A$ and $Q(I) \rhd A'$ then $A = A'$.*

Proof. By coinduction on the definition of $Q(I) \rhd A$. □

From the definition of $Q(I) \rhd A$, it follows that Q terminates with I when A is finite. Now we come to the definition of TSNI.

TSNI requires that intuitively an observer at l cannot distinguish two executions on I_1 and I_2 where $I_1|_l = I_2|_l$ if either these two executions both terminate and consume the same visible input events and generates the same visible output events, or these two executions both diverge. When they both diverge, there are no constraints on consumed input events or generated output events.

Definition IV.1 (TSNI [1])**.** *A state Q is* termination-sensitively noninterferent *(denoted by $Q \in TSNI$) if for all l, for all I_1 and I_2*

$$I_1|_l = I_2|_l \wedge \mathit{fin}(A_1) \implies \mathit{fin}(A_2) \wedge I_1'|_l = I_2'|_l \wedge O_1|_l = O_2|_l,$$

where $Q(I_1) \rhd \langle I_1', O_1\rangle$, $Q(I_2) \rhd \langle I_2', O_2\rangle$, $Q(I_1) \rhd A_1$, and $Q(I_2) \rhd A_2$.

Example IV.1 (Non-TSNI Programs)**.** *Examples of programs that are not TSNI are presented in Fig. 2, where the levels of chH and chL are respectively H and L, and $L \sqsubset H$.*

- *Program 1 does not generate any output event. However, this program is not TSNI since depending on the input*

```
1: input x from chH
2: if x > 1 then
3:     input x from chL
4: else
5:     skip
```

(a) Program 1

```
1: output 1 to chL
2: input x from chH
3: if x > 1 then
4:     while 1 do skip
5: else skip
```

(b) Program 2

Fig. 2: Non-TSNI Programs

```
1: input x from chH
2: if x > 1 then
3:     output 0 to chL
4: else
5:     output 1 to chL
6: while 1 do skip
```

(a) Program 3

```
1: input x from chH
2: if x > 1 then
3:     output 0 to chL
4: else
5:     skip
6:     output 0 to chL
```

(b) Program 4

Fig. 3: TSNI Programs

event from chH, an input event from chL may be consumed. Indeed, we consider $I_1 = chH(2) : chL(0)$ and $I_2 = chH(0) : chL(0)$. On I_1, the input consumed by the program is I_1' where $I_1'(chL) = chL(0)$. On I_2, the input consumed by the program is I_2' where $I_2'(chL) = []$. We can easily check that $I_1|_L = I_2|_L$ but $I_1'|_L \neq I_2'|_L$.

- *Program 2 always sends $chL(1)$ to chL. However, this program is not TSNI since its termination behavior depends on the input event from chH. Indeed, the program on $I_1 = chH(2)$ diverges since the while loop is executed, while it terminates on $I_2 = chH(0)$.*

Example IV.2 (TSNI Program)**.** *Examples of programs that are TSNI are presented in Fig. 3, where the levels of chH and chL are respectively H and L, and $L \sqsubset H$.*

- *Program 3 always diverges. Therefore, this program is TSNI.*
- *Program 4 is TSNI since regardless of the input events on chH, the output events generated by the program are always $chL(0)$ and the program always terminates.*

B. Indirect Termination-Sensitive Noninterference

TSNI presented in the previous section assumes that an observer at l can observe consumed input events and generated output events only when the execution terminates. However, this assumption may be violated [11]. In addition, TSNI assumes that given an execution, an observer at l can decide whether there is a step of computation or not (even in the case where there is no visible event). However, if the observer can only observe visible events to him, this assumption is too strong.

Therefore, we propose Indirect Termination-Sensitive Noninterference (ITSNI) that takes the problems mentioned above into account. Regarding ITSNI, two executions appear to be the same to an observer at l if they consume the same visible input events and generate the same visible output events at l.

Definition IV.2. *A state Q satisfies* indirect termination-sensitive noninterference *(denoted by $Q \in ITSNI$) if for all l, for all I_1 and I_2,*

$$I_1|_l = I_2|_l \implies I_1'|_l = I_2'|_l \wedge O_1|_l = O_2|_l,$$

where $Q(I_1) \rhd \langle I_1', O_1\rangle$ and $Q(I_2) \rhd \langle I_2', O_2\rangle$.

ITSNI does not imply TSNI since ITSNI does not put restrictions on termination of programs. A program that is ITSNI but not TSNI can be found in Example IV.4. TSNI does not imply ITSNI either since TSNI does not have constraints on diverging executions. A program that is TSNI but not ITSNI can be found in Example IV.3.

Example IV.3 (Non-ITSNI Program)**.** *Program 3 in Fig. 3 (which is a TSNI program) is not ITSNI since the output events sent to chL depend on the input events from chH. Program 1 in Fig. 2 is not ITSNI either since as described in Example IV.1, the consumption of an input event from chL depends on confidential input event on chH.*

Example IV.4 (ITSNI Program)**.** *We can easily check that Program 4 in Fig. 3 is ITSNI. As described in Example IV.1, Program 2 in Fig. 2 is not TSNI since the termination behavior of this program depends on the input event from chH. However, this program is ITSNI because the output event sent to chL is always $chL(1)$.*

V. Enforcement Mechanism

Indistinguishability: Intuitively, an enforcement mechanism is precise if the following holds: if the original state already satisfies the policy, then the enforcement mechanism must preserve the semantics of the original state [1], [8], [9], [12]. In other words, if the original state is good, the behavior of the state under the enforcement mechanism and the behavior of the original state are indistinguishable.

Following [12], we require that the indistinguishability relation is an equivalence relation, and if Q_1 and Q_2 are indistinguishable w.r.t. policy $\mathcal{P}$ (denoted by $Q_1 \cong^{\mathcal{P}} Q_2$), then the policy is not able to distinguish them.

$$Q_1 \cong^{\mathcal{P}} Q_2 \implies (Q_1 \in \mathcal{P} \Leftrightarrow Q_2 \in \mathcal{P})$$

In addition, we argue that indistinguishability should be defined based on what can be observed by a normal user. For TSNI, a normal user can decide whether an execution is still in progress. For terminated executions, he can also observe all input events consumed and output events generated. Hence, two states are indistinguishable w.r.t. TSNI if on any input, both of them either diverge, or converge and must consume the same input events and generate the same output events. For ITSNI, since a normal user cannot observe whether there is a step of execution or not, two states are indistinguishable w.r.t. ITSNI if on any input, they consume the same input events and generate the same output events.

The definitions of indistinguishability relations of TSNI and ITSNI are as below. In the definition, we write $Q(I) \rhd_M \langle I', O\rangle$ or $Q(I) \rhd_M A$ when Q is a state of a program whose transition relation is $\twoheadrightarrow_M$.

Definition V.1 (Indistinguishability of TSNI)**.** *Two states Q_1 and Q_2 are indistinguishable w.r.t TSNI (denoted by $Q_1 \cong^{TS} Q_2$) if for all I,*

$$\big(\mathit{fin}(A_1) \wedge \mathit{fin}(A_2) \wedge I_1' = I_2' \wedge O_1 = O_2\big) \vee \big(\mathit{inf}(A_1) \wedge \mathit{inf}(A_2)\big),$$

where $Q_1(I) \rhd_{M_1} \langle I_1', O_1\rangle$, $Q_2(I) \rhd_{M_2} \langle I_2', O_2\rangle$, $Q_1(I) \rhd_{M_1} A_1$ and $Q_2(I) \rhd_{M_2} A_2$.

Definition V.2 (Indistinguishability of ITSNI)**.** *Two states Q_1 and Q_2 are indistinguishable w.r.t ITSNI (denoted by $Q_1 \cong^{ITS} Q_2$) if for all I, it follows that $O_1 = O_2$ and $I_1' = I_2'$, where $Q_1(I) \rhd_{M_1} \langle I_1', O_1\rangle$, and $Q_2(I) \rhd_{M_2} \langle I_2', O_2\rangle$.*

We can check that $\cong^{TS}$ and $\cong^{ITS}$ are equivalence relations. As proven in Lemma V.1 and Lemma V.2, our definitions of indistinguishability relations satisfy the condition that related states cannot be distinguished by policy. Hence, $\cong^{TS}$ and $\cong^{ITS}$ satisfy all constraints for indistinguishability relation.

Lemma V.1. $Q_1 \cong^{TS} Q_2 \implies (Q_1 \in \mathit{TSNI} \Leftrightarrow Q_2 \in \mathit{TSNI})$.

Proof. From definitions of TSNI and $\cong^{TS}$. □

Lemma V.2. $Q_1 \cong^{ITS} Q_2 \Longrightarrow (Q_1 \in \mathit{ITSNI} \Leftrightarrow Q_2 \in \mathit{ITSNI})$.

Proof. From definitions of ITSNI and $\cong^{ITS}$. □

Enforcement mechanism: As in §IV, we define enforcement mechanisms on states, not on the whole programs. In the below definition, the definition of precision is based on $\cong^{\mathcal{P}}$.

Definition V.3. *An enforcement mechanism EM of a policy $\mathcal{P}$ is a total and recursive function from states to states. An enforcement mechanism is said to be* sound*, respectively* precise*, if it satisfies:*

- *Soundness:* $\forall Q, \mathit{EM}(Q) \in \mathcal{P}$,
- *Precision:* $\forall Q \in \mathcal{P}, \forall I : Q \cong^{\mathcal{P}} \mathit{EM}(Q)$.

A. Enforcement mechanism of TSNI

It is worth noting that our definition of precision for TSNI does not take into account the order of events. Furthermore, a program may generate output events before it terminates. Even with these differences, the result in Section II still applies.

Theorem V.1. *There is no sound and precise enforcement mechanism for TSNI.*

Proof. We consider a complete language that satisfies Definition III.1 s.t.: (1) there is only one channel; (2) a program in the language must consume an integer number from a channel at the beginning of the execution and it can only send an integer number to a channel right before its termination (if we use the syntax of the language presented in §III, programs are of the form **input** x **from** $ch; P'$; **output** e **to** ch, where there is no input and output command in P'); and (3) there is only one initial state. Since there is only one initial state, this initial state can be considered as a program.

We have that the restricted language described here and the language in Section II are equivalent (they are both complete, programs are deterministic, and a program consumes an integer number and may generate an integer number).

We now consider the setting where the lattice has two levels: L and H and $L \sqsubset H$, and the channel in the restricted language is at H (notice that the restricted language has only one channel). For programs in the restricted language, under our setting, TSNI is actually TS defined in Definition II.2.

Suppose that there is a sound and precise enforcement mechanism for TSNI in the complete language without any restriction. Then there must be a sound and precise enforcement mechanism for TSNI in the restricted language. Since there is a sound and precise enforcement mechanism for TSNI in the restricted language, we have a sound and precise enforcement mechanism for TSNI in the setting above. In other words, there is a sound and precise enforcement mechanism for TS.

As proven in Theorem II.1, there is no enforcement mechanism for TS. Hence, there is no sound and precise enforcement mechanism for TSNI. □

Remark V.1. *Notice that the proof of Theorem V.1 does not depend on the equality between low streams, that is the indistinguishability criterion in noninterference policies. Hence, our proof generalizes to other more general forms of noninterference where the indistiguishability criterion is based on equivalence relations like the ones presented in Section 3 of [3] thus capturing declassification policies [13], [14].*

Remark V.2. *TSNI in Definition IV.1 and in Theorem V.1 does not put any constraint on diverging executions since it assumes that an observer at l can only observe consumed input events and generated output events when executions terminate. This assumption may be violated [11]. Thus, we can strengthen TSNI by further requiring that divering executions on equivalent inputs at l must have the same visible events at l. W.r.t. this strengthened version of TSNI, Program 3 in Fig. 3 is not a good program. By using the similar reasoning in the proof of Theorem V.1, we can also prove that there is no sound and precise enforcement mechanism for the strengthened version of TSNI.*

VI. Secure Multi-Execution

Secure multi-execution (SME) was presented in [1] by Devriese and Piessens. The basic idea of SME is that SME executes several local executions (i.e. copies of the original program), each corresponding to a security level and carefully handles input events and output events of these local executions. Before presenting the semantics of SME, we introduce some auxiliary notations.

A waiting input W is a function from channels to input event queues. We abuse the notation and use $[]$ for the empty queue. A non-empty input event queue is of the form $i.K$, where i is at the head of the queue and will be consumed first. Given a queue K and an input event i, the result of appending i to K is $K.i$.

Semantics of local execution: A state of a local execution at l is of the form $\langle P, \mu, W\rangle_l$. A state $\langle P, \mu, W\rangle_l$ is a terminated state if P is **skip**. The semantics of local executions

$$\textsc{Lassign}\ \frac{v = \mu(e)}{\langle x := e, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{skip}, \mu[x \mapsto v], W\rangle_l} \qquad \textsc{Lif1}\ \frac{\mu(e) = v \qquad v = 1}{\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle P_1, \mu, W\rangle_l}$$

$$\textsc{Lif2}\ \frac{\mu(e) = v \qquad v \neq 1}{\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle P_2, \mu, W\rangle_l} \qquad \textsc{Lwhile1}\ \frac{\mu(e) = v \qquad v = 1}{\langle \textbf{while } e \textbf{ do } P, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle P; \textbf{while } e \textbf{ do } P, \mu, W\rangle_l}$$

$$\textsc{Lwhile2}\ \frac{\mu(e) = v \qquad v \neq 1}{\langle \textbf{while } e \textbf{ do } P, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{skip}, \mu, W\rangle_l} \qquad \textsc{Lseq1}\ \frac{\langle P_1, \mu, W\rangle_l \xrightarrow{a} \langle P_1', \mu', W'\rangle_l}{\langle P_1; P_2, \mu, W\rangle_l \xrightarrow{a}_E \langle P_1'; P_2, \mu', W'\rangle_l} \qquad \textsc{Lseq2}\ \frac{}{\langle \textbf{skip}; P_2, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle P_2, \mu, W\rangle_l}$$

$$\textsc{Lin1}\ \frac{\Gamma(ch) = l' \qquad l \not\sqsubseteq l'}{\langle \textbf{input } x \textbf{ from } ch, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{skip}, \mu[x \mapsto \mathit{def}(ch)], W\rangle_l} \qquad \textsc{Lin2}\ \frac{\Gamma(ch) = l}{\langle \textbf{input } x \textbf{ from } ch, \mu, W\rangle_l \xrightarrow{ch(v)}_E \langle \textbf{skip}, \mu[x \mapsto v], W\rangle_l}$$

$$\textsc{Lin3}\ \frac{\Gamma(ch) = l' \qquad l \sqsubset l' \qquad W(ch) = ch(v).K}{\langle \textbf{input } x \textbf{ from } ch, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{skip}, \mu[x \mapsto v], W[ch \mapsto K]\rangle_l} \qquad \textsc{Lin4}\ \frac{\Gamma(ch) = l' \qquad l \sqsubset l' \qquad W(ch) = []}{\langle \textbf{input } x \textbf{ from } ch, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{input } x \textbf{ from } ch, \mu, W\rangle_l}$$

$$\textsc{Lout1}\ \frac{\Gamma(ch) = l \qquad \mu(e) = v \qquad o = ch(v)}{\langle \textbf{output } e \textbf{ to } ch, \mu, W\rangle_l \xrightarrow{o}_E \langle \textbf{skip}, \mu, W\rangle_l} \qquad \textsc{Lout2}\ \frac{\Gamma(ch) \neq l \qquad \mu(e) = v}{\langle \textbf{output } e \textbf{ to } ch, \mu, W\rangle_l \xrightarrow{\circledast}_E \langle \textbf{skip}, \mu, W\rangle_l}$$

Fig. 4: Semantics of local executions

are described in Fig. 4. The semantics rules for assignment, if, while, sequential and skip commands are similar to the ones of programs. When the local execution at l generates an output event to a channel *ch* at level l, this output event is allowed (rule LOut1). Otherwise, the output event is suppressed (rule LOut2).

When the local execution at l asks for an input event on *ch* at l' where $l \not\sqsubseteq l'$, as described in rule LIn1, the execution consumes a default value returned by *def*(*ch*), where *def*() is a function that maps channels to default values (hence, the local execution does not depend on confidential data). When the local execution at l asks for an input event on *ch* also at l, the local execution consumes an input event from the environment (rule LIn2). We consider the last case where the local execution at l asks for an input event on *ch* at $l' \sqsubset l$. If its local input queue $W(ch)$ is not empty, it consumes an input event from $W(ch)$ (rule LIn3). Otherwise, it has to wait (rule LIn4).

Semantics of SME: A state of SME *lecs* is a function that maps each level l to a local execution at l. A state *lecs* is terminated when all of its local executions are terminated. Given a state $\langle P, \mu\rangle$ of a program in the language presented in §III, SME(P, μ) returns *lecs* s.t. for all l, $\mathit{lecs}(l) = \langle P, \mu, W\rangle_l$, where for all *ch*, $W(ch) = []$.

SME is equipped with a scheduler responsible for scheduling local executions. The scheduler is modeled by the function *select*() [1]. Given a non-terminated state *lecs* of SME, *select*(*lecs*) determines which local execution is executed next.

The semantics of SME is described in Fig. 5, where *lec* is the configuration of a local execution. When a local execution consumes an input event from the environment, SME distributes the input event to executions at higher levels

$$\textsc{SME1}\ \frac{\begin{array}{c}\mathit{select}(\mathit{lecs}) = l \\ \mathit{lecs}(l) = \mathit{lec} \qquad \mathit{lec} \xrightarrow{i}_E \mathit{lec}' \qquad i = ch(v) \\ \mathit{lecs}'' = \mathit{lecs}[l \mapsto \mathit{lec}'] \qquad \mathit{lecs}' = \mathit{distribute}(\mathit{lecs}'', ch(v))\end{array}}{\mathit{lecs} \xrightarrow{i}_S \mathit{lecs}'}$$

$$\textsc{SME2}\ \frac{\begin{array}{c}\mathit{select}(\mathit{lecs}) = l \\ \mathit{lecs}(l) = \mathit{lec} \qquad \mathit{lec} \xrightarrow{o}_E \mathit{lec}' \qquad \mathit{lecs}' = \mathit{lecs}[l \mapsto \mathit{lec}']\end{array}}{\mathit{lecs} \xrightarrow{o}_S \mathit{lecs}'}$$

$$\textsc{SME3}\ \frac{\mathit{select}(\mathit{lecs}) = l \qquad \mathit{lecs}(l) = \langle \textbf{skip}, \mu, W\rangle_l}{\mathit{lecs} \xrightarrow{\circledast}_S \mathit{lecs}}$$

Fig. 5: Semantics of SME

by using the *distribute*() function. For any *lecs* and $ch(v)$, $\mathit{distribute}(\mathit{lecs}, ch(v))$ returns an SME state where for any level l,

$$\mathit{distribute}(\mathit{lecs}, ch(v))(l) = \begin{cases} \langle P, \mu, W[ch \mapsto W(ch).ch(v)]\rangle_l & \text{if } \Gamma(ch) \sqsubset l, \\ & \text{where } \mathit{lecs}(l) = \langle P, \mu, W\rangle_l, \\ \mathit{lecs}(l) & \text{if } \Gamma(ch) \not\sqsubset l. \end{cases}$$

When a local execution generates an output event, SME will also generate this event (rule SME2). When a terminated local execution is selected to be executed, since the local execution has terminated, SME just generates a silent output event (rule SME3). Notice that depending on the definition of *select*(), rule SME3 may not be applied. For example, in the case of low-prio scheduler, this rule is not applied. However, in the case of round-robin scheduler, this rule is used.

[1] We consider only deterministic programs and SME is a program. To guarantee that SME is deterministic, we consider only deterministic schedulers. Therefore, we model schedulers by using a function.

Lemma VI.1. *For any lecs, lecs satisfies Definition III.1.*

Proof. We prove this lemma by case analysis on the semantics rules of SME. Important facts used in the proof are that *select*(), *distribute*(), and *def*() are functions. □

Since *lecs* satisfies Definition III.1, we can write $\mathit{lecs}(I) \rhd_{SME} \langle I', O\rangle$ and $\mathit{lecs}(I) \rhd_{SME} A$. Because the semantics relation for *lecs* is clear (i.e. $\rightarrow_S$), we write $\mathit{lecs}(I) \rhd \langle I', O\rangle$ and $\mathit{lecs}(I) \rhd A$ instead of $\mathit{lecs}(I) \rhd_{SME} \langle I', O\rangle$ and $\mathit{lecs}(I) \rhd_{SME} A$.

Next we investigate SME with different schedulers. First we look at the low-prio scheduler [1] since it is used in the original SME paper [1]. We give an example demonstrating that SME with a low-prio scheduler cannot enforce soundly TSNI when the lattice is totally ordered. We show that SME with the low-prio scheduler cannot enforce precisely ITSNI and it offers ITSNI only when lattice is totally ordered. Therefore, we consider fair schedulers [15] and we prove that with a fair scheduler, SME can enforce soundly and precisely ITSNI.

A. Low-prio scheduler

SME with a low-prio scheduler is denoted by SMEL. As presented in [1], the low-prio scheduler always chooses the non-terminated local execution at the smallest level to execute first. When the order in the lattice is not total, it will be converted to a total order and the smallest level is determined by the total order.

1) Termination sensitive noninterference: In [1], the authors proved that SMEL could enforce soundly and precisely TSNI defined with totally ordered lattice. However, their proof is not correct since as proven in Theorem V.1, TSNI cannot be enforced soundly and precisely. Indeed, SMEL does not enforce soundly TSNI even when the lattice is totally ordered as illustrated in Example VI.1.

Example VI.1 (SMEL is not TSNI sound)**.** *We consider the totally ordered lattice with two levels: L and H, where $L \sqsubset H$. Let us look at the below program where chH is a channel at level H. This program is not TSNI since the confidential data from chH influence the termination behavior of the program.*

1: **input** x **from** chH
2: **while** $x > 0$ **do skip**

Suppose that the default value for chH is 0. Since the default value for chH is 0, the local execution at L always terminates and the local execution at H always consumes an input value from chH. When the input value from chH is 0, the high execution terminates and hence, the whole SMEL terminates. When the input value from chH is 1, the high execution diverges, and hence, the whole SMEL diverges.

Since there is no channel at level L, input streams $chH(0)$ and $chH(1)$ are equivalent at L. However, SMEL on these inputs has different termination behaviors. Thus, SMEL does not enforce soundly TSNI.

In [1], the authors argued informally that we can extend a finite lattice to a totally ordered one, and since SMEL with the scheduler defined based on the totally ordered lattice was TSNI sound and precise, SMEL was also TSNI sound and precise for the arbitrary lattice. This argument is not correct since from Theorem V.1, there is no sound and precise enforcement mechanism for TSNI. The argument is flawed since by extending a lattice to a totally ordered one, we introduce new and unwanted flows. For example, by extending the lattice with four levels L, M_1, M_2, H, where $L \sqsubset M_i \sqsubset H$, to $L \sqsubset_T M_1 \sqsubset_T M_2 \sqsubset_T H$, we introduce a new and unwanted flow: the flow from M_1 to M_2. In addition, as explained above, SMEL is not TSNI sound when the lattice is total.

2) Indirect termination sensitive noninterference: For non-total lattices, SMEL is not ITSNI since as illustrated in Example VI.2, the execution at a level l may influence the execution at l' where l and l' are incomparable levels.

Example VI.2 (Non-totally ordered lattice)**.** *When the order $\sqsubset$ in the lattice is not total, SMEL does not offer ITSNI. We consider the lattice with four levels: L, M_1, M_2, and H, where $L \sqsubset M_1 \sqsubset H$ and $L \sqsubset M_2 \sqsubset H$. The program below has two channels: chM_1 and chM_2 at respectively levels M_1 and M_2. The default value for chM_1 is 0.*

1: **input** x **from** chM_1
2: **while** $x > 1$ **do skip**
3: **input** y **from** chM_2

As described above, when the order $\sqsubset$ is not total, a total order $\sqsubset_T$ is constructed and the smallest level is determined based on the constructed total order. Suppose that the total order is $L \sqsubset_T M_1 \sqsubset_T M_2 \sqsubset_T H$. W.r.t. this total order, the execution at M_2 starts executing only when the execution at M_1 terminates. However, depending on the input event from chM_1, the execution at M_1 may diverge and hence, the execution at M_2 may be starved.

Let us look at $I_1 = I[chM_1 \mapsto chM_1(2), chM_2 \mapsto chM_2(1)]$ and $I_2 = I[chM_1 \mapsto chM_1(1), chM_2 \mapsto chM_2(1)]$ for some I. W.r.t. I_1, the local execution at M_1 loops forever and no input event from chM_2 is consumed. W.r.t. I_2, the local execution at M_1 terminates and the execution at M_2 can consume $chM_2(1)$ (notice that because the default event for chM_1 is $chM_1(0)$, the execution at M_2 does not go to the loop). We have that $I_1|_{M_2} = I_2|_{M_2}$ but $I_1'|_{M_2} \neq I_2'|_{M_2}$, where $\mathit{lecs}(I_1) \rhd \langle I_1', O_1\rangle$, $\mathit{lecs}(I_2) \rhd \langle I_2', O_2\rangle$, and lecs is the SMEL state constructed with the program. In other words, when the lattice is not total, SMEL does not offer ITSNI.

As illustrated in Example VI.2, SMEL cannot enforce soundly ITSNI because of the influence of local executions at incomparable levels. Therefore hereafter in this sub-section we consider only lattices with total orders.

W.r.t. the low-prio scheduler, we have the following property which states that if $\mathit{lecs}(l)$ is selected, it must be not terminated and all local executions at lower levels must be terminated.

Property VI.1. *For any non-terminated lecs,*

$$\mathit{select}(\mathit{lecs}) = l \implies \neg \mathit{terminated}(\mathit{lecs}(l)) \wedge (\forall l' : l' \sqsubset l \implies \mathit{terminated}(\mathit{lecs}(l'))).$$

When the local execution at l is selected to be executed next, from Property VI.1, all local executions at lower levels have terminated. Hence, there are no consumed input events or generated output events on channels at levels smaller than l. Therefore, we have the following lemma.

Lemma VI.1. *Suppose that select($lecs$) $= l$. It follows that for all I and ch s.t. $\Gamma(ch) \sqsubset l$, it follows that $I'(ch) = O(ch) = []$, where $Q(I) \triangleright \langle I', O\rangle$.*

Proof. By coinduction and case analysis. □

We next prove that if SMEL may still consume or generate events at l or smaller, on equivalent inputs, the events visible at l are the same.

Lemma VI.2. *Suppose that select($lecs$) $= l'$ where $l' \sqsubseteq l$. For any I_1 and I_2 s.t. $I_1|_l = I_2|_l$, it follows that $I_1'|_l = I_2'|_l$ and $O_1|_l = O_2|_l$, where $lecs(I_1) \triangleright \langle I_1', O_1\rangle$ and $lecs(I_2) \triangleright \langle I_2', O_2\rangle$.*

Proof. By coinduction and case analysis. □

Theorem VI.1. *For any lecs of SMEL, lecs is ITSNI.*

Proof. Let *lecs* be an arbitrary state of SMEL. Let I_1 and I_2 be arbitrary inputs s.t. $I_1|_l = I_2|_l$. If *terminated*(*lecs*), we can check that the theorem holds. We consider the case where $\neg$*terminated*(*lecs*). Let $l' =$ *select*(*lecs*). We have two sub cases. In the first case, $l' \not\sqsubseteq l$. Since the order is total, $l \sqsubset l'$. From Lemma VI.1, for all *ch* s.t. $\Gamma(ch) \sqsubseteq l$, we have that $I_1'(ch) = I_2'(ch) = O_1(ch) = O_2(ch) = []$. Therefore, $I_1'|_l = I_2'|_l$ and $O_1|_l = O_2|_l$. In the second case, $l' \sqsubseteq l$. From Lemma VI.2, $I_1'|_l = I_2'|_l$ and $O_1|_l = O_2|_l$. □

Corollary VI.1. *For any $\langle P, \mu\rangle$, SMEL($\langle P, \mu\rangle$) is ITSNI.*

From Corollary VI.1, SMEL is sound w.r.t. ITSNI. Unfortunately, SMEL is not precise w.r.t. ITSNI. The reason is that the high execution may starve and hence, there is no input events consumed or no output events generated on high channels. An example illustrating this case is described in Example VI.3

Example VI.3 (SMEL not precise w.r.t. ITSNI)**.** *Suppose that the lattice has two levels: $L \sqsubset H$ and the level of chH is H. We consider the following program.*

1: **output** 1 **to** chH
2: **while** 1 **do skip**

This program only sends a fixed output event to the high channel and it does not generate any visible output at L. This program is ITSNI. When we run this program with SMEL, the execution at L is scheduled to run first and its output is suppressed by SMEL. Since the execution at L runs forever, the execution at H starves and hence, there is no output event sent to chH by SMEL. In other words, SMEL is not precise w.r.t. ITSNI.

B. Fair scheduler

SME with a fair scheduler [15] is denoted by SMEF. W.r.t. a fair scheduler, no local execution starves. We have the following property for SMEF.

Property VI.2. *For any lecs and l,*

$$lecs(l) = \langle P, \mu, W\rangle_l \ \wedge\ P \neq \textbf{skip} \implies \exists lecs'', lecs' : lecs \twoheadrightarrow_S^* lecs'' \twoheadrightarrow_S lecs' \wedge select(lecs'') = l \wedge lecs''(l) = \langle P, \mu, W'\rangle_l,$$

where $\twoheadrightarrow_S^*$ is the transitive and reflexive closure of $\twoheadrightarrow_S$.

We next establish the relation between the input consumed and the output generated by SMEF and the corresponding ones by local executions. In order to do so, given *lecs* and I, we define $\langle\!\langle lecs, I\rangle\!\rangle$ that constructs a function from levels to tuples of the form $\langle I', O\rangle$. Intuitively, if $\langle\!\langle lecs, I\rangle\!\rangle(l) = \langle I', O\rangle$, then the local execution $lecs(l)$ consumes the input I' and generates the output O. Since the local execution can only consume directly input events at level l, and can only generate output event at l, for $\langle\!\langle lecs, I\rangle\!\rangle(l) = \langle I', O\rangle$, we have that I' and O have only events at l.

In order to define $\langle\!\langle lecs, I\rangle\!\rangle$, we define auxiliary notations. We define $I\,\|_l$ that returns the input with streams on channels at l, i.e. $dom(I\,\|_l) = \{ch \in dom(I) | \Gamma(ch) = l\}$ and for any $ch \in dom(I\,\|_l)$, $I\,\|_l\,(ch) = I(ch)$. Similarly, we have $O\,\|_l$.

To reason about the behavior of a local execution, we need to know which input events are consumed by executions at lower levels. Since a local execution may consume an infinite number of input events and the waiting input of a local execution can only accept finite queues, we define lecx, an extended version of state of local execution. An extended state of a local execution is also of the form $\langle P, \mu, W\rangle_l$. Different from the (normal) state of a local execution, W in a lecx is a mapping from channels to streams. The semantics for lecx at level l is the same as the one of local execution (e.g. if it needs an input event on a channel at l, it will consume one from the environment, if it needs an input event on a channel at $l' \sqsubset l$, it either consumes one from its waiting input or it has to wait). We have that lecx satisfies Definition III.1, hence, we can write $\mathsf{lecx}(I) \triangleright \langle I', O\rangle$.

The inductive definition of $\langle\!\langle lecs, I\rangle\!\rangle$ is as below, where $\bot$ is the smallest level of the lattice, $lecs(l).P$, $lecs(l).\mu$, and $lecs(l).W$ are respectively the program, the memory, and the waiting input in $lecs(l)$, and $[\![l, \langle\!\langle lecs, I\rangle\!\rangle]\!]$ constructs a waiting input for the local execution lecx at l by combining the current waiting input with the inputs consumed by lower executions. Here, to simplify the presentation, we abuse ":" and use it to concatenate a finite queue to a stream.

- if $\mathsf{lecx}(I\,\|_\bot) \triangleright \langle I', O\rangle$, where $\mathsf{lecx} = lecs(\bot)$, then $\langle\!\langle lecs, I\rangle\!\rangle(\bot) = \langle I', O\rangle$,
- if $\mathsf{lecx}(I\,\|_l) \triangleright \langle I', O\rangle$, then $\langle\!\langle lecs, I\rangle\!\rangle(l) = \langle I', O\rangle$, where $\mathsf{lecx} = \langle lecs(l).P, lecs(l).\mu, [\![l, \langle\!\langle lecs, I\rangle\!\rangle]\!]\rangle_l$, and

$$[\![l, \langle\!\langle lecs, I\rangle\!\rangle]\!](ch) \triangleq \begin{cases} lecs(l).W(ch) : I'(ch) & \text{if } \Gamma(ch) \sqsubset l, \\ & \quad \text{where } \langle\!\langle lecs, I\rangle\!\rangle(\Gamma(ch)) = \langle I', O\rangle, \\ lecs(l).W(ch) & \text{if } \Gamma(ch) \not\sqsubset l. \end{cases}$$

Next we prove that input events consumed and output events generated by SMEF at level l are the same as the corresponding ones by the local execution at l.

Lemma VI.3. *For any lecs and I, suppose that* $lecs(I) \rhd \langle I', O\rangle$*, and for any* l*,* $\langle\!\langle lecs, I\rangle\!\rangle(l) = \langle I'', O'\rangle$*. For any* l*, it follows that* $I'' = I' \|_l$ *and* $O' = O \|_l$*.*

Proof. The proof is in Appendix. □

We now can prove that SMEF is ITSNI.

Theorem VI.2 (Soundness)**.** *For any lecs, lecs is ITSNI.*

Proof. The proof is in Appendix. □

Corollary VI.2. *For any* $\langle P, \mu\rangle$*, SMEF*$(\langle P, \mu\rangle)$ *is ITSNI.*

We now prove that SMEF is a precise enforcement mechanism of ITSNI.

Theorem VI.3 (Precision)**.** *SMEF is precise w.r.t. ITSNI.*

Proof. Let Q be the state that is ITSNI, $Q(I) \rhd \langle I', O\rangle$ and *SMEF*$(Q)(I) \rhd \langle I'', O'\rangle$.

We define $\lceil I\rceil^l$ that returns an input where for any *ch*, if *ch* is at levels $l' \not\sqsubseteq l$, it is mapped to D_{ch}, a stream of default input events for *ch* (that is any input event in D_{ch} is *ch*(*def*(*ch*))).

$$\lceil I\rceil^l(ch) = \begin{cases} I(ch) & \text{if } \Gamma(ch) \sqsubseteq l, \\ D_{ch} & \text{if } \Gamma(ch) \not\sqsubseteq l. \end{cases}$$

We write *lecs* for *SMEF*(Q). For any l, we have that $lecs(l) = Q$. Let $lecs(I) \rhd \langle I'', O'\rangle$ and for any l, $Q(\lceil I\rceil^l) \rhd \langle I'_{1l}, O_{1l}\rangle$, and $\mathsf{lecx}_l(I\|_l) \rhd \langle I'_l, O_l\rangle$.

We prove $I'_{1l}\|_l = I'_l$ and $O_{1l}\|_l = O_l$, $I'|_l = I''|_l$ and $O|_l = O''|_l$ by induction on l.

Base case: $l = \bot$. Let $\mathsf{lecx}_\bot(I\ \|_\bot) \rhd \langle I'_\bot, O_\bot\rangle$ and $Q(\lceil I\rceil^\bot) \rhd \langle I'_{1\bot}, O_{1\bot}\rangle$. We have that $\mathsf{lecx} = lecs(\bot) = Q$. We also have that $\lceil I\rceil^\bot\|_\bot = I\|_\bot$. From the semantics of original program and the semantics of local execution, we have that $I'_{1\bot}\|_\bot = I'_\bot$, and $O_{1\bot}\|_\bot = O_\bot$.

Since Q is ITSNI, and $I|_\bot = \lceil I\rceil^\bot|_\bot$, we have that $I'|_\bot = I'_1|_\bot$ and $O|_\bot = O_1|_\bot$ (where $Q(I) \rhd \langle I', O\rangle$ and $Q(\lceil I\rceil^\bot) \rhd \langle I'_1, O_1\rangle$). From Lemma VI.3, $I''\|_\bot = I'_\bot$ and $O'\|_\bot = O_\bot$.

Therefore, we have that $I''\ \|_\bot = I'_\bot$, $I'_{1\bot}\ \|_\bot = I'_\bot$, and $I'|_\bot = I'_{1\bot}|_\bot$ and hence, $I'\|_\bot = I'_{1\bot}\|_\bot$. Thus, $I'\|_\bot = I''\|_\bot$ and hence, $I'|_\bot = I''|_\bot$. Similarly, $O|_\bot = O'|_\bot$.

IH: Suppose that the two statements holds for all $l' \sqsubset l$. We now consider l. From the semantics of SME, the input provided to the execution at l is $I_1 = I\|_l$. The waiting input for this execution is $W = [\![l, \langle\!\langle lecs, I\rangle\!\rangle]\!]$. Let $\mathsf{lecx}_l(I_1) \rhd \langle I'_l, O_l\rangle$, and $Q(\lceil I\rceil^l) \rhd \langle I'_{1l}, O_{1l}\rangle$. We now prove that $I'_1\|_l = I'_l$ and $O_1\|_l = O_l$.

For the sake of contradiction, suppose that $I'_1\|_l \neq I'_l$ or $O_1\|_l \neq O_l$. We consider the case where $I'_1\|_l \neq I'_l$ (the proof for the remaining case is similar). Since $I'_1\|_l \neq I'_l$, there exists I_0 (which has only input events at l) and I_a s.t. $I_a\|_l = I_0$, $Q \xrightarrow{I_a} Q'$ and $\mathsf{lecx}_l \xrightarrow{I_0}_E \mathsf{lecx}'_l$, and Q' and lecx' consume different input events at l, where $Q \xrightarrow{I_a} Q'$ mean that Q consumes I_a and goes to Q'. We claim that $Q'.P = \mathsf{lecx}'.P$ and $Q'.\mu = \mathsf{lecx}'.\mu$, where $Q'.P$ is the program in Q' and $Q'.\mu$ is the memory in Q'. Suppose that $Q'.P \neq \mathsf{lecx}'.P$ or $Q'.\mu \neq \mathsf{lecx}'.\mu$. Since the programs and memories in Q and lecx are the same, they must consume a different input event at $l' \sqsubset l$. Thus, we have that $I_{l'} \neq I_{1l}\|_{l'}$. Since Q is ITSNI, we have that $I_{l'} \neq I_{1l'}\|_{l'}$. Contradiction. Therefore, we consider Q' and lecx' s.t. $Q'.P = \mathsf{lecx}'.P$ and $Q'.\mu = \mathsf{lecx}'.\mu$. Hence, Q' and lecx' must consume the same input event at l. Contradiction.

Since Q is ITSNI, and $I|_l = \lceil I\rceil^l|_l$, we have that $I'|_l = I'_{1l}|_l$ and $O|_l = O_{1l}|_l$, where $Q(I) \rhd \langle I', O\rangle$ and $Q(\lceil I\rceil^l) \rhd \langle I'_{1l}, O_{1l}\rangle$. From Lemma VI.3, $I''\|_l = I'_{l'}$ and $O'\|_l = O_{l'}$.

Therefore, we have that $I''\|_l = I'_l$, $I'_{1l}\|_l = I'_l$, and $I'|_l = I'_{1l}|_l$ and hence $I'\|_l = I'_{1l}\|_l$. Thus, $I'\|_l = I''\|_l$. From IH, for any $l' \sqsubset l$, $I'|_{l'} = I''|_{l'}$. Hence, it follows that $I'|_l = I''|_l$. Similarly, $O|_l = O'|_l$. □

VII. A TYPE SYSTEM FOR ITSNI

The goal of this section is to show that ITSNI can be statically enforced. We define an enforcement mechanism based on a type and effect system [16] which is ITSNI sound. In this section, we overload security environments to denote mappings from channels and variables to security levels, that is Γ : *Channel* $\cup$ $\mathbf{Var} \mapsto \mathcal{L}$. We assume that security environments are total functions. We overload $\sqcup$ as a binary operation between security environments such that $(\Gamma_1 \sqcup \Gamma_2)(x)$ is equal to $\Gamma_1(x) \sqcup \Gamma_2(x)$ for all $x \in \mathbf{Var}$ and $(\Gamma_1 \sqcup \Gamma_2)(ch)$ is equal to $\Gamma_1(ch) \sqcup \Gamma_2(ch)$ for all $ch \in$ *Channel*. Let $\Gamma_1 \sqsubseteq \Gamma_2$ hold if $\Gamma_1(ch) = \Gamma_2(ch)$ for all $ch \in$ *Channel* and $\Gamma_1(x) \sqsubseteq \Gamma_2(x)$ for all $x \in \mathbf{Var}$.

The intuition behind the type system is based on ideas from works on typing concurrent languages [17]: visible outputs are not allowed after loops with guards depending on non visible inputs. For a given security environment Γ, typing judgements $\Gamma \vdash e : \tau$ mean that expression e reads variables of level at most τ. Typing judgements for commands, written $\Gamma, \tau \vdash c : \tau, \eta$ mean that for "termination" level τ, that is the security level on which termination currently depends [17], c is secure, writes variables of level at least τ, and has effect η. An effect a pair (Γ', τ') where Γ' is a security environment and τ' is a security level called termination effect. The intuition is that Γ' is a security environment to sequence, and termination effect τ' records the new level on which termination depends.

The rules for our type and effect system, listed in Fig. 6, are mostly standard [2], [18], [19] except for: the propagation of effects and rules In, Out, and While:

Input commands are typable by Rule In: their security level is bound to the security level of the channel ch (τ_x) and the security level of the channel ch is assigned as the new security level of variable x in the effect of the command. The security level of the channel is bound by the termination level.

Rule While binds τ_e as the security level of the guard of the while command, and τ_P as the level of the body of the while command. Since body P needs to be typed with a single security environment, we require its security environment to

$$\textsc{Skip}\ \frac{}{\Gamma, \tau \vdash \textbf{skip}\ : \top, (\Gamma, \tau)}$$

$$\textsc{Assign}\ \frac{\Gamma(x) = \tau_x \qquad \Gamma \vdash e : \tau_e \qquad \tau_e \sqsubseteq \tau_x \qquad \tau \sqsubseteq \tau_x}{\Gamma, \tau \vdash x := e\ : \tau_x, (\Gamma, \tau)}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau_e \qquad \tau_e \sqsubseteq \tau_1 \sqcap \tau_2 \\ \Gamma, \tau \vdash P_1\ : \tau_1, (\Gamma_1, \tau_1') \qquad \Gamma, \tau \vdash P_2\ : \tau_2, (\Gamma_2, \tau_2')\end{array}}{\Gamma, \tau \vdash \textbf{if}\ e\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ : \tau_1 \sqcap \tau_2, (\Gamma_1 \sqcup \Gamma_2, \tau_1' \sqcup \tau_2')}$$

$$\textsc{While}\ \frac{\begin{array}{c}\Gamma_P \vdash e : \tau_e \\ \Gamma_P, \tau_P \vdash P\ : \tau_P', (\Gamma_P, \tau_P) \qquad \Gamma \sqsubseteq \Gamma_P \qquad \tau \sqsubseteq \tau_P \qquad \tau_e \sqsubseteq \tau_P'\end{array}}{\Gamma, \tau \vdash \textbf{while}\ e\ \textbf{do}\ P\ : \tau_P', (\Gamma_P, \tau_e \sqcup \tau_P)}$$

$$\textsc{In}\ \frac{\Gamma(ch) = \tau_x \qquad \tau \sqsubseteq \tau_x}{\Gamma, \tau \vdash \textbf{input}\ x\ \textbf{from}\ ch\ : \tau_x, (\Gamma[x \mapsto \tau_x], \tau)}$$

$$\textsc{Out}\ \frac{\tau \sqsubseteq \Gamma(ch) \qquad \Gamma \vdash e : \tau_e \qquad \tau_e \sqsubseteq \Gamma(ch)}{\Gamma, \tau \vdash \textbf{output}\ e\ \textbf{to}\ ch\ : \Gamma(ch), (\Gamma, \tau)}$$

$$\textsc{Seq}\ \frac{\Gamma, \tau \vdash c_1\ : \tau_1, (\Gamma', \tau') \qquad \Gamma', \tau' \vdash c_2\ : \tau_2, \eta''}{\Gamma, \tau \vdash c_1; c_2\ : \tau_1 \sqcap \tau_2, \eta''}$$

Fig. 6: Type system for ITSNI in programs with I/O operations

be the same as the security environment in its effect (Γ_P). Moreover, the contraint $\Gamma \sqsubseteq \Gamma_P$ enforces that channels in Γ and Γ_P have the same security level and enforces variables in Γ_P which are affected by P are not downgraded. The constraint $\tau_e \sqsubseteq \tau_P'$ records the implicit flow from the guard to the body of the command, as in Rule If. Finally, the new termination effect $\tau_e \sqcup \tau_P'$ records that termination may depend on this while command: if the guard of the loop is higher than the current termination effect τ_P then the termination effect also raises.

Notice that this makes the type system not precise since examples of ITSNI programs as the following will not be typable because the termination effect of the while loop will be higher than the channel to output:

1: **input** x **from** chH
2: **while** x **do** $x := 0$
3: **output** x **to** chL

Notice that the type system could be made more precise by adding a typing rule that does not change the termination effect of a while command if one can prove as a hypothesis that the while commands terminate for all memories and inputs (this idea is developed in [20], [21] in the context of different security policies).

Rule Out for typing outputs is the key to obtain ITSNI. In particular, the constraint $\tau \sqsubseteq \Gamma(ch)$ constrains the security level of outputs on a channel to be lower bound by the current termination level (τ). Finally, type τ_e binds the security level of the expression and $\tau_e \sqsubseteq \Gamma(ch)$ records the explicit flow from the expression to the output channel, similar to the constraint in Rule Assign.

Definition VII.1 (Typability of P)**.** *Program P is typable with environment Γ, denoted $\Gamma \vdash P$, if there exists τ, τ', η such that $\Gamma, \tau \vdash c\ : \tau', \eta$.*

Theorem VII.1 (Soundness)**.** *If $\Gamma \vdash P$ then P is ITSNI.*

Proof. The proof is in Appendix. □

VIII. Related Work

TS security definitions: TSNI was first formally defined by Volpano and Smith [22]. Other formulations of termination sensitive information flow have followed. Abadi et al. and Sabelfeld and Sands [23], [24] formalize TS security by partial equivalence relations, representing degrees of security, in the deterministic and nondeterministic setting respectively. Askarov et al. [11] propose a definition of TSNI (later called progress-sensitive noninterference [25]) on a simple language where programs do not consume input and can only generate output. The idea behind their TSNI is similar to the idea behind ITSNI: their TSNI is defined based on events visible to attacker and it is independent from actual terminations of executions. Bohannon et al. [10] present several notions of security in the context of general reactive programs: programs that alternate between computing and interacting with one or more external agents. They propose CPCT-Security as an adaptation of TSNI in the context of reactive languages. They also propose a weaker version called CP, which is analogous to ITSNI in the context of general reactive programs. They do not discuss differences in their enforcement: while CPCT-security can be enforced soundly and precisely, CP cannot. Stronger definitions such as timing sensitive noninterference [26] have been proposed. In timing sensitive security, the assumption is that an attacker can observe the time of computation. This is known as timing channels. Timing channels are particularly important when enforcing security for concurrent program [27], [28]. Observational determinism is an information flow policy proposed with the goal of generalizing noninterference to capture timing channels in concurrent programs. The observational determinism policy resembles to our ITSNI policy. Observational determinism differs from ITSNI because ITSNI does not consider equivalences between traces up to stuttering.

TS (non precise) static mechanisms: Several sound, but not precise, TS enforcement mechanisms exist. Volpano and Smith [22], propose a static TSNI enforcement mechanism which is sound but not precise. Their enforcement is based on a type system that accepts programs without while loops whose guard depends on confidential information. Agat and Sands [29] study how algorithms for searching and sorting can (efficiently) be made timing sensitive noninterferent when working on collections of confidential data. Terauchi [30] proposes a type system to enforce observational determinism in the context of a concurrent language. The type system proposed by Boudol and Castellani [28] in the context of a concurrent language is the closest to our proposal of Section 7. In their type system, they close timing channels by avoiding low assignments after high loops. We reuse this idea to enforce

ITSNI in our setting. The soundness property that they enforce is a form of termination sensitive noninterference defined as a bisimulation. Barthe et al. [3] propose to model check termination sensitive information flow policies (generalisations of noninterference that can capture declassification) characterised as CTL and LTL formulas using the self composition technique. Recently, Antonopoulos [31] present a technique (to replace self composition) called decomposition to enforce general information flow properties and in particular avoid timing sensitive non interference (in a non precise way).

Sound and precise (non computable) TSNI methods: Barthe et al. [3] propose the verification of termination sensitive generalised noninterference using the weakest precondition (wp) transformer with total correctness. Since the wp transformer is sound and complete, it follows that the verification is sound and precise. This does not contradict our result since wp is not an enforcement mechanism because it is not computable (case of while). Bielova and Rezk [5], [6] model and compare different dynamic mechanism for noninterference including a non-computable version of SME, relying on an oracle that can decide the halting problem, which is sound and complete for TSNI.

TS dynamic enforcements: The first proposal of SME [1] (that we present in Section VI) was claimed sound for TSNI and Timing Sensitive noninterference using a low priority scheduler. Several variants of secure multi execution based mechanisms have been proposed since then (e.g. [32], [33]) but most of them are proved sound for termination insensitive security [11]. We discuss only TS cases. Barthe et al. [4] propose a dynamic enforcement mechanism based on static transformation of programs. They claim the mechanism to be TSNI sound and precise using the TSNI claim of SME soundness and precision [1]. Unfortunately, since this claim does not hold (as we explain in Section VI), the mistake propagates to their result. Kashyap et al. [34] point out that SME [1] does not enforce soundly a version of TSNI when the security lattice is not totally ordered (their argument is similar to our explanation in Example VI.2). Notice that the version of TSNI in their argument resembles ITSNI but it is specific to SME as it relies on observations of local executions of SME. They also propose different schedulers for different TS policies. In contrast to our work, they do not consider precision. Rafnsson and Sabelfeld [35] prove that SME can enforce soundly timing-sensitive and progressive-sensitive noninterference when the scheduler is fair and deterministic. They prove that SME can enforce precisely progressive-sensitive non-interference when the scheduler is *high lead* (e.g a round robin scheduler is a high lead scheduler). Notice that different from the scheduler used in our SMEF, their schedulers are required to be independent from input events consumed and behaviour of local executions. Their technique for full-transparency can be used to further improve the precision of SMEF. Our results on ITSNI sound and precise enforcement mechanisms can be extended to existent variants of SME that can handle declassification policies [32], [33], [35]. Ngo et al. [9] propose general sound and precise enforcement mechanisms for hyper properties [36] (hyper properties are generalisation of properties and are defined in terms of several executions. For example, noninterference is a hyperproperty defined in term of 2 executions.). Their sound and precise enforcements do no contradict our result because they work only in the setting of terminating programs.

General results on enforceable properties: The Rice theorem [7], [37] says that non-trivial program properties (properties which are either empty or always true) are not decidable. Notice that an enforcement mechanism does not need to decide the property in order to enforce it soundly and precisely. An example is SME for ITSNI, as shown in this work. Hamlen et al. [8] characterize classes of security policies which are enforceable (by Rice Theorem, not decidable). In particular, they present the secret file policy. According to our understanding, this policy is termination sensitive noninterference: programs behaviour should be the same with or without confidential information, here a secret file. They informally explain that the secret file policy can be enforced soundly and precisely. However, this is not a contradiction to our main result because the notion of precision that they use is weaker than ours: the precision definition used for the secret file policy says that normal user can only observe termination but not any (other) output. Indeed, their precision only requires that if the program satisfies TSNI and on an input, the original program terminates with an output O_1, then the enforcement mechanism on the same input terminates with O_2, and O_1 and O_2 may be different. In Section V we discuss that a good notion of precision should reflect what a normal user can observe: we can argue that if termination is the only output that a normal user can observe, computation might not be very useful.

IX. Conclusion

The main conclusion of our work is that termination sensitivity of information flow policies is a subtle property, and it makes sense to distinguish two notions of termination sensitivity:

- a notion that talks about actual termination of programs (TS),
- and a notion that formalizes termination as it can be observed indirectly (ITS).

Depending on attacker models and/or the security objective that one wishes to achieve, both styles of policies are useful. However, another important conclusion from this paper is that ITS policies are easier to enforce than TS policies: we have shown that ITS policies can be enforced soundly and precisely and TS polices can not.

As future work, we plan to study if our results also hold in the case of nondeterministic (and more generally probabilistic) programs.

REFERENCES

[1] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. of IEEE SP 2010*, ser. SP '10, 2010, pp. 109–124.
[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 2003.
[3] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.
[4] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas, "Secure multi-execution through static program transformation," in *Proc. of FMOODS 2012 and FORTE 2012*, 2012, pp. 186–202.
[5] N. Bielova and T. Rezk, "A taxonomy of information flow monitors," in *Proc. of POST 2016*, 2016, pp. 46–67.
[6] ——, "Spot the difference: Secure multi-execution and multiple facets," in *Proc. of ESORICS 2016*, 2016, pp. 501–519.
[7] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
[8] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 175–205, Jan. 2006.
[9] M. Ngo, F. Massacci, D. Milushev, and F. Piessens, "Runtime enforcement of security policies on black box reactive programs," in *Proc. of POPL 2015*, ser. POPL '15, 2015, pp. 43–54.
[10] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proc. of CCS 2009*, ser. CCS '09, 2009, pp. 79–90.
[11] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proc. of ESORICS 2008*, ser. ESORICS '08, 2008, pp. 333–348.
[12] J. Ligatti, L. Bauer, and D. Walker, "Enforcing non-safety security policies with program monitors," in *Proc. of ESORICS 2005*, ser. ESORICS'05, 2005, pp. 355–373.
[13] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.
[14] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *Proc. of CSF 2015*, 2015.
[15] A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts*, 8th ed. Wiley Publishing, 2011.
[16] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Proc. of POPL 1988*, ser. POPL '88, 1988.
[17] G. Boudol and I. Castellani, "Noninterference for concurrent programs and thread systems," *Theoretical Computer Science*, 2002.
[18] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comp. Sec.*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996.
[19] G. Barthe, T. Rezk, and D. A. Naumann, "Deriving an information flow checker and certifying compiler for java," in *Proc. of IEEE SP 2006*, ser. SP '06, 2006, pp. 230–242.
[20] G. Boudol, "On typing information flow," in *Proc. of ICTAC 2005*, 2005.
[21] G. Barthe, S. Cavadini, and T. Rezk, "Tractable enforcement of declassification policies," in *Proc. of CSF 2008*, 2008.
[22] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proc. of CSFW 1997*, ser. CSFW '97, 1997.
[23] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proc. of POPL 1999*, 1999, pp. 147–160.
[24] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher-Order and Symbolic Computation*, vol. 14, no. 1, pp. 59–91, 2001.
[25] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *Proc. of CCS 2012*, ser. CCS '12, 2012, pp. 881–893.
[26] J. Agat, "Transforming out timing leaks," in *Proc. of POPL 2000*, ser. POPL '00, 2000.
[27] D. M. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *Journal of Computer Security*, vol. 7, no. 1, 1999.
[28] G. Boudol and I. Castellani, "Noninterference for concurrent programs," in *Proc. of ICALP 2001*, 2001, pp. 382–395.
[29] J. Agat and D. Sands, "On confidentiality and algorithms," in *Proc. of IEEE SP 2001*, 2001, pp. 64–77.
[30] T. Terauchi, "A type system for observational determinism," in *Proc. of CSF 2008*, 2008, pp. 287–300.
[31] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *Proc. of PLDI*, 2017, pp. 362–375.
[32] M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk, "Stateful declassification policies for event-driven programs," in *Proc. of CSF 2014*, ser. CSF' 14, 2014, pp. 293–307.
[33] I. Bolosteanu and D. Garg, "Asymmetric secure multi-execution with declassification," in *Proc. of POST 2016*, 2016, pp. 24–45.
[34] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and termination-sensitive secure information flow: Exploring a new approach," in *Proc. of IEEE SP 2011*, ser. SP '11, 2011, pp. 413–428.
[35] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," *Journal of Computer Security*, vol. 24, no. 1, pp. 39–90, 2016.
[36] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.
[37] A. Asperti and C. Armentano, "A page in number theory," *J. Formalized Reasoning*, vol. 1, no. 1, pp. 1–23, 2008.

X. PROOF OF SECTION 6

Lemma VI.3. *For any lecs and I, suppose that $lecs(I) \rhd \langle I', O\rangle$, and for any l, $\langle\!\langle lecs, I\rangle\!\rangle(l) = \langle I'', O'\rangle$. For any l, it follows that $I'' = I' \|_l$ and $O' = O \|_l$.*

Proof. Base case: $l = \bot$. Let $\langle\!\langle lecs, I\rangle\!\rangle(\bot) = \langle I'', O'\rangle$, and $\mathsf{lecx} = lecs(\bot)$. We need to prove that $I'' = I' \|_\bot$ and $O' = O \|_\bot$.

For the sake of contradiction, we assume that $I'' \neq I' \|_\bot$ or $O' \neq O \|_\bot$.

Case 1: $I'' \neq I' \|_\bot$. Since $I'' \neq I' \|_\bot$, there exists I_0 (which may be an empty stream that has only input events from $\bot$) s.t. $lecs(\bot) \xrightarrow{I_0}_E lec'$, and $\mathsf{lecx} \xrightarrow{I_0}_E \mathsf{lecx}'$, and lec' and lecx' will consume an input event at $\bot$. Since $lecs(\bot) = \mathsf{lecx}$, since they consume the same input events at $\bot$, since programs are deterministic, we have that $lec'.P = \mathsf{lecx}.P$ and $lec'.\mu = \mathsf{lecx}'.\mu$. Since $I'' \neq I' \|_\bot$, we have the following cases:

- lec' is stuck because of the scheduler and $\mathsf{lecx}' \xrightarrow{i_1}_E \mathsf{lecx}''$. Contradiction since the scheduler is fair.
- $lec' \xrightarrow{i_1}_E lec''$, $\mathsf{lecx}' \xrightarrow{i_2}_E \mathsf{lecx}''$, and $i_1 \neq i_2$. Contradiction since lec' and lecx' have the same programs, memories, and inputs at $\bot$.

Case 2: $O' \neq O \|_l$. As proven above, $I'' = I' \|_\bot$. Since the inputs consumed are the same, local executions are deterministic, and the scheduler is fair, we have that $O' = O \|_l$. Contradiction.

IH: Suppose that the lemma holds for all $l' \sqsubset l$. That is for all $l' \sqsubset l$, we have that $I'_{l'} = I' \|_{l'}$ and $O_{l'} = O \|_{l'}$, where $\langle\!\langle lecs, I\rangle\!\rangle(l') = \langle I'_{l'}, O_{l'}\rangle$.

We now need to prove that $I'' = I' \|_l$ and $O' = O \|_l$, where $\langle\!\langle lecs, I\rangle\!\rangle(l') = \langle I'', O'\rangle$.

Let $\mathsf{lecx} = \langle lecs(l).P, lecs(l).\mu, [\![l, \langle\!\langle lecs, I\rangle\!\rangle]\!]\rangle_l$. As for the case of $\bot$, for the sake of contradiction, we suppose that $I'' \neq I' \|_l$ or $O' \neq O \|_l$.

Case 1: $I'' \neq I' \|_l$. Since $I'' \neq I' \|_l$, there exists I_0 which may be an empty stream and has only input events at l s.t. $lecs(l) \xrightarrow{I_0}_E lec'$, $\mathsf{lecx} \xrightarrow{I_0}_E \mathsf{lecx}'$, and lec' and lecx' will consume an input event at l and the input events consumed by them makes $I'' \neq I' \|_l$.

We now prove that $lec'.P = \mathsf{lecx}'.P$ and $lec'.\mu = \mathsf{lecx}'.\mu$. From the construction of lecx, we have that $\mathsf{lecx}.P = lecs(l).P$, $\mathsf{lecx}.\mu = lecs(l).\mu$, the waiting input for lecx is

from the waiting input of $lecs(l)$ and inputs consumed by lower executions. Since program are deterministic, $lec'.P \neq \mathsf{lecx}'.P$ or $lec'.\mu \neq \mathsf{lecx}'.\mu$ only when $lecs(l)$ and lecx consume a different input event. Since they consume the same input events at l (that is I_0), they must consume a different input event at $l' \sqsubset l$. The input events at l' available for the local execution at l' is $lecs(l).W(ch) : I'(ch)$, where $\Gamma(ch) = l'$. The input events at l' available for lecx is $lecs(l).W(ch) : I'_{l'}(ch)$, where $\Gamma(ch) = l'$. Since $lecs(l)$ and lecx consume a different input event at $l' \sqsubset l$, there exists ch s.t. $\Gamma(ch) = l'$ and $lecs(l).W(ch) : I'(ch) \neq lecs(l).W(ch) : I'_{l'}(ch)$. In other words, there exists ch s.t. $\Gamma(ch) = l'$ and $I'(ch) \neq I'_{l'}$. Therefore, $I' \|_{l'} \neq I'_{l'}$. Contradiction.

We have the following cases:

- lec' is stuck forever because of the scheduler and $\mathsf{lecx}' \xrightarrow{i_1}_E \mathsf{lecx}''$. Contradiction since the scheduler is fair.
- $lec' \xrightarrow{i_1}_E lec''$, $\mathsf{lecx}' \xrightarrow{i_2}_E \mathsf{lecx}''$, and $i_1 \neq i_2$. Contradiction since lec' and lecx' have the same programs, memories, and inputs at l (that is the input obtained by removing I_0 from $I \|_l$).

Case 2: $O' \neq O \|_l$. As proven above, $I'' = I' \|_\perp$. Since the inputs consumed are the same, local executions are deterministic, and the scheduler is fair, we have that $O' = O \|_l$. Contradiction. □

Theorem VI.2 (Soundness)**.** *For any lecs, lecs is ITSNI.*

Proof. Let $lecs(I_1) \rhd \langle I'_1, O_1 \rangle$ and $lecs(I_2) \rhd \langle I'_2, O_2 \rangle$. From Lemma VI.3, for any l', we have that: $I'_i \|_{l'} = I''_{il'}$ and $O_i \|_{l'} = O''_{il'}$, where $\langle\!\langle lecs, I_i \rangle\!\rangle(l') = \langle I''_{il'}, O'_{il'} \rangle$.

For any $l' \sqsubseteq l$, we have that $I_1 \|_{l'} = I_2 \|_{l'}$. Thus, for $l' \sqsubseteq l$, we have that $I''_{1l'} = I''_{2l'}$ and $O'_{1l'} = O'_{2l'}$. Therefore, for $l' \sqsubseteq l$, $I'_1 \|_{l'} = I'_2 \|_{l'}$ and $O_1 \|_{l'} = O_2 \|_{l'}$. Thus, $I'_1|_l = I'_2|_l$ and $O_1|_l = O_2|_l$. Hence, SMEF is ITSNI. □

XI. Proof of Section 7

Given a security level l, program P is typable as high with environment Γ, denoted $\Gamma \vdash P : H_l$, if there exists τ, τ', η such that $\Gamma, \tau \vdash c \ : \tau', \eta$ and $\tau' \not\sqsubseteq l$. We write $\Gamma \vdash P : H$ when l is clear from the context. Given a security level l, program P is typable as low with environment Γ, denoted $\Gamma \vdash P : L_l$, if there exists τ, τ', η such that $\Gamma, \tau \vdash c \ : \tau', \eta$ and $\tau' \sqsubseteq l$. We write $\Gamma \vdash P : L$ when l is clear from the context.

Auxiliary definitions for the proof: We define $\xrightarrow{*}$ as the reflexive and transitive closure of $\rightarrow$. We also define $\xrightarrow{I,O}$ as the reflexive and transitive closure of $\rightarrow$ where I_1 and O_2 recall the input streams and output streams used for the transitions. In this section, we coinductively extend the notion of visible event and overload $visible()$ for streams.

We require the definition of concatenation of streams, denoted $++$, defined coinductively as follows:

$$\frac{}{[] ++ [] = []} \qquad \frac{}{[] ++ S = S} \qquad \frac{s ++ S' = S''}{s : S ++ S' = s : S''}$$

We overload $++$ to be applied to input and outputs of our language, in a per channel basis. We define $\Gamma \vdash \mu =_l \mu'$ if $\mu(x) = \mu'(x)$ for all variables x st $\Gamma(x) \sqsubseteq l$. We define equality with respect to a security level l between events, denoted $=_l$, as:

$$\frac{\neg visible_l(a_1) \quad \neg visible_l(a_1)}{a_1 =_l a_2} \qquad \frac{a_1 = a_2 \quad visible_l(a_1) \quad visible_l(a_1)}{a_1 =_l a_2}$$

For simplicity, in the following we will assume that there is a strict separation between input and output channels. This simplifies greatly the following definition and lifting this restriction does not shed any light on the main result. We will use the following function on input streams for the bisimulation:

$$g(a, I, ch) = \begin{cases} I & \text{if } I = [] \text{ or } a \neq ch(v) \text{ or } ch \text{ is not an input channel,} \\ I' & \text{if } a = ch(v) \text{ and } ch \text{ is an input channel} \\ & \quad \text{and } I = a : I' \end{cases}$$

$$f(I_1, I, ch) = \begin{cases} I & I_1 = [] \\ f(I'_1, g(a, I), ch) & I_1 = a : I'_1 \end{cases}$$

$\delta(I_1, I) = \{ch \mapsto f(I_1(ch), I(ch), ch) | ch \in dom(I)\}$. We assume that $I_1(ch) = []$ if $ch \notin dom(I_1)$.

Intuitively, $\delta(I_1, I)$ is the tail of input stream I after consuming stream I_1.

The proof of the theorem requires a coinduction technique since the policy to prove talks about infinite streams. We rely on the following bisimulation for the proof:

Definition XI.1 (Bisimulation)**.** *A $\mathcal{B}_l$ is a symmetric relation on programs such that*
$\langle I_1, P_1, \mu_1, \Gamma_1 \rangle \mathcal{B}_l \langle I_2, P_2, \mu_2, \Gamma_2 \rangle$ *implies that* $\Gamma_1 \vdash \mu_1 =_l \mu_2 \wedge \Gamma_2 \vdash \mu_1 =_l \mu_2 \wedge l|_{I_1} = l|_{I_2}$ *and one of the following holds:*

$$\begin{array}{l} (1) \langle P_1, \mu_1 \rangle \xrightarrow{a_1} \langle P'_1, \mu'_1 \rangle \wedge visible_l(a_1) \implies \\ \langle P_2, \mu_2 \rangle \xrightarrow{a_2} \langle P'_2, \mu'_2 \rangle \wedge a_1 =_l a_2 \wedge \\ \langle \delta([a_1], I_1), P'_1, \mu'_1, \Gamma'_1 \rangle \mathcal{B}_l \langle \delta([a_2], I_2), P'_2, \mu'_2, \Gamma'_2 \rangle \wedge \\ \exists I'_1, I'_2, O_1, O_2 . P_i, \mu_i(I_i) \rhd \langle I'_i, O_i \rangle \\ (2) \langle P_1, \mu_1 \rangle \xrightarrow{I'_1, O_1} \langle P'_1, \mu'_1 \rangle \wedge \neg visible_l(I'_1, O_1) \implies \\ \langle P_2, \mu_2 \rangle \xrightarrow{I'_2, O_2} \langle P'_2, \mu'_2 \rangle \wedge \neg visible_l(I'_2, O_2)) \wedge \\ \langle \delta(I'_1, I_1), P'_1, \mu'_1, \Gamma'_1 \rangle \mathcal{B}_l \langle \delta(I'_2, I_2), P'_2, \mu'_2, \Gamma'_2 \rangle \wedge \\ \exists I'_1, I'_2, O_1, O_2 . P_i, \mu_i(I_i) \rhd \langle I'_i, O_i \rangle \end{array}$$

Bisimulation $\mathcal{B}_l$ defines a family of bisimulations. Let $\sim$ be the largest of all bisimulations.

A. Auxiliary lemmas

We define lemmas HIGH and LOW (for the cases of programs typable as high and low respectively) to state the invariants that are preserved for typable programs after one step of execution.

Lemma XI.1 (High)**.** *Let l be a security level, μ be a memory, and I an input. If $\Gamma, \tau \vdash P \ : \tau', (\Gamma', \tau'')$ and $\tau' \not\sqsubseteq l$ then*

1) *if $\langle P, \mu \rangle \xrightarrow{a} \langle P', \mu' \rangle$ then $\neg visible_l(a)$*
2) *if $\langle P, \mu \rangle \xrightarrow{a} \langle P', \mu' \rangle$ then $\Gamma' \vdash \mu =_l \mu'$*
3) *$\tau \sqsubseteq \tau''$*

4) *if* $\langle P, \mu\rangle \xrightarrow{a} \langle P', \mu'\rangle$ *then* $\exists \Gamma_1, \tau_1, \tau_2, \eta. \Gamma' \sqsubseteq \Gamma_1 \wedge \tau'' \sqsubseteq \tau_1$. $\Gamma_1, \tau_1 \vdash P' : \tau_2, \eta$ *and* $\tau_2 \not\sqsubseteq l$
5) *for all variable* x *such that* $\Gamma(x) \sqsubseteq l$ *then* $\Gamma'(x) = \Gamma(x)$.

Proof. By structural induction on P.

Case **skip**: By typing $\tau'' = \tau$ and (3) holds and $\Gamma = \Gamma'$ and (5) holds.

Case $x := e$: By semantics $\langle x := e, \mu\rangle \xrightarrow{\circledast} \langle \textbf{skip}, \mu[x \mapsto v]\rangle$ so (1) is satisfied. Since by typing $\Gamma(x) = \tau'$ and $\tau = \tau''$, and by hypothesis $\tau' \not\sqsubseteq$, then (2) and (3) are satisfied. Item (4) is satisfied because **skip** is always typable with any environments. Finally $\Gamma = \Gamma'$ for this case so (5) is satisfied.

Case **input** x **from** ch: by semantics $\langle \textbf{input } x \textbf{ from } ch, \mu\rangle \xrightarrow{ch(v)} \langle \textbf{skip}, \mu[x \mapsto v]\rangle$. By hypothesis $\tau' \neq l$ and by typing $\Gamma(ch) = \tau'$. Hence, since $\neg \mathit{visible}_l(ch)$ holds, item (1) is satisfied. By typing $\tau' = \Gamma'(x)$. Hence μ_i and $\mu_i[x \mapsto v_i]$ are equal in all variables x st $\Gamma'(x) \sqsubseteq l$. Since μ_1 and μ_2 are equal in all variables x st $\Gamma(x) \sqsubseteq l$ and by typing Γ and Γ' are equal except for x, it follows that $\mu_1[x \mapsto v_1]$ and $\mu_2[x \mapsto v_2]$ are equal in all variables x st $\Gamma'(x) \sqsubseteq l$ and item (2) is satisfied. By typing $\tau'' = \tau$ so item (3) is satisfied. Item (4) is satisfied because **skip** is always typable with any environments. By typing Γ and Γ' are equal except for x whose type is $\tau'' \not\sqsubseteq l$. Hence item (5) is satisfied.

Case **output** e **to** ch: By semantics we have $\langle \textbf{output } e \textbf{ to } ch, \mu_i\rangle \xrightarrow{ch(\mu(e))} \langle \textbf{skip}, \mu_i\rangle$. By typing, $\Gamma(ch) = \tau'$ and by hypothesis $\tau' \not\sqsubseteq$ so $\neg \mathit{visible}_l(ch)$ of item (2) holds. Item (2) holds trivially since memory does not change for this step. Item (3) holds because $\tau = \tau''$. Item (4) is satisfied because **skip** is always typable with any environments. Item (5) holds trivially because $\Gamma = \Gamma'$ for this case.

Case **if** e **then** P_1 **else** P_2: By semantics, we have $\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu\rangle \xrightarrow{\circledast} \langle P_1, \mu\rangle$. Item (1) holds since $\neg \mathit{visible}_l(\circledast)$ holds. Item (2) holds trivially since memories do not change in this step and $=_l$ is reflexive. By typing $\tau'' = \tau$ so item (3) holds. By hypotheses of if rule in typing, P_1 is typable using Γ and τ. Thus item (4) holds. Finally $\Gamma = \Gamma'$ so item (5) holds. The case for $\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu\rangle \xrightarrow{\circledast} \langle P_2, \mu\rangle$ is symmetric.

Case **while** e **do** P_1: By semantics we have two cases.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu\rangle \xrightarrow{\circledast} \langle P_1; \textbf{while } e \textbf{ do } P_1, \mu\rangle$ Item (1) holds since $\neg \mathit{visible}_l(\circledast)$ holds. Item (2) holds trivially since memories do not change in this step and $=_l$ is reflexive. By typing $\tau'' = \tau \sqcup \tau_e \sqcup \tau_P$ then if $\tau \not\sqsubseteq l$ then $\tau'' \not\sqsubseteq l$. Hence item (3) holds. By hypothesis, we have that $\Gamma, \tau \vdash \textbf{while } e \textbf{ do } P_1 : \tau', (\Gamma', \tau \sqcup \tau_e \sqcup \tau_P)$ By typing hypotheses of while we have that (a) $\Gamma_P \vdash e : \tau_e$ and (b) $\Gamma', \tau_P \vdash P_1 : \tau', (\Gamma', \tau_P)$ and (c) $\Gamma \sqsubseteq \Gamma'$ and (d) $\tau \sqsubseteq \tau_P$ and (e) $\tau_e \sqsubseteq \tau'$. In order to type $P_1; \textbf{while } e \textbf{ do } P_1$ using rule SEQ, it is enough to show that the typing judgment $\Gamma', \tau_P \vdash \textbf{while } e \textbf{ do } P_1 : \tau', (\Gamma', \tau \sqcup \tau_e \sqcup \tau_P)$ holds. We show that each hypothesis of the while typing rule is satisfied: By (a) we have that $\Gamma_P \vdash e : \tau_e$. By (b) we have $\Gamma', \tau_P \vdash P_1 : \tau', (\Gamma', \tau_P)$. The third hypothesis of the typing rule is trivially satisfied since we are using the same environment in the hypothesis and the conclusion. By (c) the last hypothesis is satisfied. Hence (4) holds. Finally, by inductive hypothesis on P_1 (5) holds: we can apply the hypothesis because by (b) P_1 is typable with τ' and $\tau' \not\sqsubseteq l$ by lemma hypothesis.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu\rangle \xrightarrow{\circledast} \langle \textbf{skip}, \mu\rangle$ Item (1) holds since $\neg \mathit{visible}_l(\circledast)$ holds. Item (2) holds trivially since memories do not change in this step and $=_l$ is reflexive. By typing $\tau'' = \tau \sqcup \tau_e \sqcup \tau_P$ then if $\tau \not\sqsubseteq l$ then $\tau'' \not\sqsubseteq l$. Hence item (3) holds. Item (4) is satisfied because **skip** is always typable with any environments. By inductive hypothesis on P_1 (5) holds.

Case $P_1; P_2$. By the SEQ rule $\exists \Gamma'', \tau'', \eta, \Gamma''', \tau_1, \eta'$ such that $\Gamma'', \tau''' \vdash P_1 : \tau_1, \eta$ and $\Gamma''', \tau_1 \vdash P_2 : \tau_2, \eta'$ where $\tau' = \tau_1 \sqcap \tau_2$. Since by hypothesis $\tau' \not\sqsubseteq l$ and $\tau_1 \sqcap \tau_2 \sqsubseteq \tau_1, \tau_2$ we conclude that $\tau_1 \not\sqsubseteq l$ and $\tau_2 \not\sqsubseteq l$. Thus, we can apply inductive hypothesis on P_1 and P_2 to conclude. □

Lemma XI.2 (Monotonically increasing termination level)**.** *Let* $\Gamma, \tau \vdash P : \tau', (\Gamma', \tau'')$. *Then* $\tau \leq \tau'$ *and* $\tau \leq \tau''$.

Proof. By induction on the length of the typing derivation tree. □

Lemma XI.3 (Low-1Step)**.** *Let* l *be a security level and* μ_1, μ_2 *be memories such that* $\Gamma \vdash \mu_1 =_l \mu_2$. *If* $\Gamma, \tau \vdash P : \tau', (\Gamma', \tau'')$ *and* $\tau' \sqsubseteq l$ *and* $\langle P, \mu_1\rangle \xrightarrow{a_1} \langle P_1, \mu_1\rangle$ *then*

1) $\langle P, \mu_2\rangle \xrightarrow{a_2} \langle P_2, \mu_2'\rangle$ *and* $\Gamma' \vdash \mu_1' =_l \mu_2'$ *and* $a_1 =_l a_2$
2) $\exists \tau_1, \tau_2, \tau_1'', \tau_2''$ *such that* $\Gamma', \tau'' \vdash P_i : \tau_i', (\Gamma_i', \tau_i'')$ *and either one of the following options holds:*
   a) $P_1 = P_2$
   b) $P_1 = P_1'; P' \wedge P_2 = P_2'; P' \wedge P_1' \neq P_2'$ *and* $\exists \tau_1', \tau_2', \tau_1''', \tau_2'''$ *s.t.* $\Gamma_i', \tau_i'' \vdash P_i' : \tau_i''', (\Gamma_i''', \tau_i''') \wedge \tau_i'' \not\sqsubseteq l$

where $\langle P, \mu_i\rangle(I_i) \rhd \langle I_i', O_i'\rangle$ *and* $I_1|_l = I_2|_l$.

Proof. By structural induction on P. We prove this lemma by structural induction on P.

Case $x := e$: By semantics $\langle x := e, \mu_i\rangle \xrightarrow{\circledast} \langle \textbf{skip}, \mu_i[x \mapsto \mu_i(e)]\rangle$. To see that item (1) holds we have to prove that $\circledast = \circledast$ and $\Gamma \vdash \mu_1[x \mapsto \mu_1(e)] =_l \mu_2[x \mapsto \mu_2(e)]$. The first statement is trivially true. For the second statement notice that $\mu_i(e)$ depends only on low variables of memory μ_i because of the typing constraint $\tau_e \sqsubseteq \tau'$ and the hypothesis that $\tau' \sqsubseteq l$ and the fact that τ_e is an upper bound of all levels of variables occurring in e. Hence, because the language is deterministic and hypothesis $\Gamma \vdash \mu_1 =_l \mu_2$ we know that $\mu_1(e) = \mu_2(e)$. Thus $\Gamma \vdash \mu_1[x \mapsto \mu_1(e)] =_l \mu_2[x \mapsto \mu_2(e)]$. Item (2.a) is satisfied because $P_i = \textbf{skip}$.

Case **input** x **from** ch: By semantics $\langle \textbf{input } x \textbf{ from } ch, \mu_i\rangle \xrightarrow{ch(v_i))} \langle \textbf{skip}, \mu_i[x \mapsto v_i]\rangle$. By typing hypothesis $\tau' = \Gamma(ch)$ and by hypothesis $\Gamma(ch) \sqsubseteq l$. Hence, since by hypothesis $I_1 =_l I_2$, we know that $v_1 = v_2$. Hence, since $\Gamma \vdash \mu_1 =_l \mu_2$, we have $\Gamma \vdash \mu_1[x \mapsto v_1] =_l \mu_2[x \mapsto v_2]$ and item (1) holds. Item (2.a) is satisfied because $P_1 = P_2 = \textbf{skip}$.

Case **output** e **to** ch: By semantics we have $\langle \textbf{output } e \textbf{ to } ch, \mu_i\rangle \overset{ch(\mu_i(e))}{\rightarrow} \langle \textbf{skip}, \mu_i\rangle$. To see that item (1) holds we have to prove that $ch(\mu_1(e)) =_l ch(\mu_2(e))$ and $\Gamma \vdash \mu_1 =_l \mu_2$. The second statement holds by lemma hypothesis. For the first statement we consider two cases. If $\Gamma(ch) \sqsubseteq l$ then we need to show that $\mu_1(e) = \mu_2(e)$. This follows from the typing constraint $\tau_e \sqsubseteq \tau'$ and the hypothesis that $\tau' \sqsubseteq l$ and the fact that τ_e is an upper bound of all levels of variables occurring in e. We will show that the subcase $\Gamma(ch) \not\sqsubseteq l$ is impossible: by typing rule OUT $\tau' = \Gamma(ch)$. But this contradicts the hypothesis that $\tau' \sqsubseteq l$. Item (2.a) is satisfied because $P_1 = P_2 =$ **skip** .

Case **if** e **then** P_1 **else** P_2: By semantics, we have $\langle \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \mu_i\rangle \overset{\circledast}{\rightarrow} \langle P_{j_i}, \mu_i\rangle$ with $j_1, j_2 \in \{1, 2\}$. Item (1) holds because $\circledast = \circledast$ holds trivially and $\Gamma \vdash \mu_1 =_l \mu_2$ holds by hypothesis. Typability of P_{j_i} follows by the typing hypotheses of rule IF: $\Gamma, \tau \vdash P_i : \tau_i'', (\Gamma_i'', \tau_i''')$.

If $\mu_1(e) = \mu_2(e)$ then because the language is deterministic and by semantics we have $P_{j_1} = P_{j_2}$ (option (2.a) of the lemma hypotheses) and we conclude. We will show that since the if is typable as low $\mu_1(e) \neq \mu_2(e)$ is not an option. Without loss of generality assume that $P_{j_1} = P_1$ and $P_{j_2} = P_2$. Suppose $\mu_1(e) \neq \mu_2(e)$. Since the language is deterministic and μ_1, μ_2 are equal in the low variables $\Gamma \vdash \mu_1 =_l \mu_2$, it is necessary the case that e depends on some high variable in order to have $\mu_1(e) \neq \mu_2(e)$. Hence, by typability of e (hypothesis of the typing rule IF), we have that $\tau_e \not\sqsubseteq l$. Hence, by typing rule constraint $\tau_e \sqsubseteq \tau_1'' \sqcap \tau_2''$ and $\tau' = \tau_1'' \sqcap \tau_2''$ (where τ' is the type of $P =$ **if** e **then** P_1 **else** P_2) , we conclude that $\tau_i'' \not\sqsubseteq l$ and $\tau' \not\sqsubseteq l$. But this is a contradiction since by lemma hypothesis $\tau' \sqsubseteq l$.

Case **while** e **do** P_1: By semantics we have four cases to analyze. For all the cases item (1) holds trivially since memories do not change and they are equivalent by hypothesis, and $\circledast = \circledast$. We focus on item (2) for each subcase.

By lemma hypothesis $\Gamma, \tau \vdash$ **while** e **do** $P_1 : \tau', (\Gamma', \tau_e \sqcup \tau_P)$ then y typing hypotheses of while we have that:
(a) $\Gamma_P \vdash e : \tau_e$ and
(b) $\Gamma', \tau_P \vdash P_1 : \tau', (\Gamma', \tau_P)$ and
(c) $\Gamma \sqsubseteq \Gamma'$ and
(d) $\tau \sqsubseteq \tau_P$ and
(e) $\tau_e \sqsubseteq \tau'$.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu_1\rangle \overset{\circledast}{\rightarrow} \langle P_1; \textbf{while } e \textbf{ do } P_1, \mu_1\rangle$ and $\langle \textbf{while } e \textbf{ do } P_1, \mu_2\rangle \overset{\circledast}{\rightarrow} \langle \textbf{skip}, \mu_2\rangle$. We will prove that this case is impossible. By semantics $\mu_1(e) \neq \mu_2(e)$. Since the language is deterministic and μ_1, μ_2 are equal in the low variables $\Gamma \vdash \mu_1 =_l \mu_2$, it is necessary the case that e depends on some high variable ($\not\sqsubseteq l$) in order to have $\mu_1(e) \neq \mu_2(e)$. Hence, by typability of e, hypothesis (a) of typing, we have that $\tau_e \not\sqsubseteq l$ since τ_e is an upper bound of all variables read in e. Hence, by typing rule constraint $\tau_e \sqsubseteq \tau'$ and $\tau' = \tau_1'' \sqcap \tau_2''$ (where τ' is the type of $P =$ **if** e **then** P_1 **else** P_2) , we conclude that $\tau_i'' \not\sqsubseteq l$ and $\tau' \not\sqsubseteq l$. But this is a contradiction since by lemma hypothesis $\tau' \sqsubseteq l$.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu_2\rangle \overset{\circledast}{\rightarrow} \langle P_1; \textbf{while } e \textbf{ do } P_1, \mu_2\rangle$ and $\langle \textbf{while } e \textbf{ do } P_1, \mu_1\rangle \overset{\circledast}{\rightarrow} \langle \textbf{skip}, \mu_1\rangle$. Symmetric to the previous case.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu_1\rangle \overset{\circledast}{\rightarrow} \langle P_1; \textbf{while } e \textbf{ do } P_1, \mu_1\rangle$ and $\langle \textbf{while } e \textbf{ do } P_1, \mu_2\rangle \overset{\circledast}{\rightarrow} \langle P_1; \textbf{while } e \textbf{ do } P_1, \mu_2\rangle$. We are in option (2a) of the conclusion of the lemma. We will prove that $\Gamma', \tau_e \sqcup \tau_P \vdash P_1; \textbf{while } e \textbf{ do } P_1 : \tau', (\Gamma', \tau_e \sqcup \tau_P)$. In order to type $P_1; \textbf{while } e \textbf{ do } P_1$ using rule SEQ, it is enough to prove that $\Gamma', \tau_e \sqcup \tau_P \vdash P_1 \tau', (\Gamma', \tau_e \sqcup \tau_P)$ and $\Gamma', \tau_e \sqcup \tau_P \vdash$ **while** e **do** $P_1 \vdash \tau', (\Gamma', \tau_e \sqcup \tau_P)$.

We first prove $\Gamma', \tau_e \sqcup \tau_P \vdash P_1 \tau', (\Gamma', \tau_e \sqcup \tau_P)$: by hypothesis (a) we have $\Gamma', \tau_P \vdash P_1 : \tau', (\Gamma', \tau_P)$. Suppose that P_1 is not typable with termination effect $\tau_P \sqcup \tau_e$. Hence, $\tau_e \not\leq \tau_P$. Then there must be a rule where the constraint on the termination level fails. Without loss of generality, say it is the output rule so that the termination level is not less or equal than the channel level $\tau_P \sqcup \tau_e \not\sqsubseteq \Gamma(ch)$. But then the type τ' of P_1 has to be at most $\Gamma(ch)$ (or smaller). Without loss of generality, say it is $\Gamma(ch)$. But by constraint (e) $\tau_e \sqsubseteq \tau'$, thus $\tau_e \sqsubseteq \Gamma(ch)$. Two cases follows:
1. $\tau_P \leq \tau_e$: then $\tau_P \sqcup \tau_e = \tau_e$ and we reach a contradiction since we have $\tau_P \sqcup \tau_e \not\sqsubseteq \Gamma(ch)$ and $\tau_e \sqsubseteq \Gamma(ch)$.
2. $\tau_P \not\leq \tau_e$ and $\tau_e \not\leq \tau_P$: by hypothesis (e) $\tau_e \sqsubseteq \tau'$ and by Lemma XI.2 we have that $\tau_P \leq \tau'$ since τ_P is the termination level of hypothesis (b). Then $\tau_P \sqcup \tau_e \leq \tau'$. Hence $\tau_P \sqcup \tau_e \leq \Gamma(ch)$. This is a contradiction since P_1 is not typable because $\tau_P \sqcup \tau_e \not\sqsubseteq \Gamma(ch)$. Hence $\Gamma', \tau_e \sqcup \tau_P \vdash P_1 \tau', (\Gamma', \tau_e \sqcup \tau_P)$

We now prove: $\Gamma', \tau_e \sqcup \tau_P \vdash$ **while** e **do** $P_1 \vdash \tau', (\Gamma', \tau_e \sqcup \tau_P)$. it is enough to show that the typing judgment $\Gamma', \tau_P \sqcup \tau_e \vdash$ **while** e **do** $P_1 : \tau', (\Gamma', \tau \sqcup \tau_e \sqcup \tau_P)$ holds. We show that each hypothesis of the while typing rule is satisfied: By (a) we have that $\Gamma_P \vdash e : \tau_e$. By the previous case, we have $\Gamma', \tau_P \sqcup \tau_e \vdash P_1 : \tau', (\Gamma', \tau_P)$. The third hypothesis of the typing rule is trivially satisfied since we are using the same environment in the hypothesis and the conclusion. By (c) the last hypothesis is satisfied, so we conclude.

Subcase $\langle \textbf{while } e \textbf{ do } P_1, \mu_1\rangle \overset{\circledast}{\rightarrow} \langle \textbf{skip}, \mu_1\rangle$ and $\langle \textbf{while } e \textbf{ do } P_1, \mu_2\rangle \overset{\circledast}{\rightarrow} \langle \textbf{skip}, \mu_2\rangle$

Program **skip** is always typable and we are in option 1 of item 2, so we conclude.

Case $P_1; P_2$. By the SEQ rule $\exists \Gamma'', \tau'', \eta, \Gamma''', \tau_1, \eta'$ such that $\Gamma'', \tau'' \vdash P_1 : \tau_1, (\Gamma''', \tau_3)$ and $\Gamma''', \tau_3 \vdash P_2 : \tau_2, \eta'$ where $\tau' = \tau_1 \sqcap \tau_2$. There are two cases to analyze.

Subcase $\tau_1 \sqsubseteq l$ We can apply inductive hypothesis on P_1 to conclude.

Subcase $\tau_1 \not\sqsubseteq l$ We prove this case by structural induction on P_1. Base cases follow directly from Lemma HIGH. Subsubcase $P_1 =$ **while** e **do** P_1' Since P_1 is typable as high we can have $\mu_1(e) \neq \mu_2(e)$. Assume that $\mu_1(e) = 1$ and $\mu_2(e) \neq 1$. Then by semantics $\langle P1; P2, \mu_1\rangle \overset{\circledast}{\rightarrow} \langle P_1'; \textbf{while } e \textbf{ do } P_1'; P_2, \mu_1\rangle$ $\langle P1; P2, \mu_2\rangle \overset{\circledast}{\rightarrow} \langle \textbf{skip}; P_2, \mu_1\rangle$ Hence we are in case (2.b) of the lemma and we have to see that **skip** and $P_1'; \textbf{while } e \textbf{ do } P_1'$ are typable as high. Program **skip** is always typable as high and $P_1'; \textbf{while } e \textbf{ do } P_1'$ is typable

as high because P_1 is, using a reasoning similar to case while of this proof.

Subsubcase $P_1 =$ **if** e **then** P_1' **else** Similar to previous case. □

Lemma XI.4 (HIGH EFFECT)**.** *Let* $\Gamma, \tau \vdash P; P'$ *such that* $\Gamma, \tau \vdash P \ : \tau', (\Gamma', \tau'')$ *and* $\tau'' \not\sqsubseteq l$*. Then* $\exists \Gamma', \tau', \tau'', \eta'$ *st* $\Gamma', \tau' \vdash P' \ : \tau'', \eta'$ *and* $\tau'' \sqsubseteq l$

Proof. By induction on length of the typing derivation. □

Proposition XI.1 ($\mathcal{H}_l$)**.** *Let* $\Gamma_1 \vdash P_1$ *and* $\Gamma_2 \vdash P_2$ *be typable as high, and let* $\mathcal{H}_l$ *be the relation consisting of pairs* $(\langle I_1, P_1, \mu_1, \Gamma_1\rangle, \langle I_2, P_2, \mu_2, \Gamma_2\rangle)$ *such that* $\Gamma_1 \vdash \mu_1 =_l \mu_2 \wedge \Gamma_2 \vdash \mu_1 =_l \mu_2 \wedge l|_{I_1} = l|_{I_2} \wedge \exists I_1', I_2', O_1, O_2. P_i, \mu_i(I_i) \rhd \langle I_i', O_i\rangle$*. Then* $\mathcal{H}_l$ *is a* $\mathcal{B}_l$ *relation.*

Proof. Let $(\langle I_1, P_1, \mu_1, \Gamma_1\rangle, \langle I_2, P_2, \mu_2, \Gamma_2\rangle)$ and $\langle P, \mu\rangle \xrightarrow{a} \langle P', \mu'\rangle$. Then by Lemma HIGH $\neg$*visible*$_l(a)$ and P' is typable by high. For the same reasons, this can be simulated by $\langle P_2, \mu_2\rangle$ (case (3)) of $\mathcal{B}_l$. Then $\mathcal{H}_l$ is a $\mathcal{B}_l$. □

B. Main results

Lemma XI.5. *If* $\Gamma \vdash P$ *then* $\langle I_1, P, \mu_1, \Gamma\rangle \sim \langle I_2, P, \mu_2, \Gamma\rangle$ *for* $\Gamma \vdash \mu_1 =_l \mu_2$ *and* $I_1|_l = I_2|_l$*.*

Proof. We define inductively the relation $\mathcal{T}_l$ as follows: A relation $\mathcal{T}_l$ is a symmetric relation on process such that $\langle I_1, P_1, \mu_1, \Gamma_1\rangle \mathcal{T}_l \langle I_2, P_2, \mu_2, \Gamma_2\rangle$ implies $\exists \tau_1', \Gamma_1', \tau_1'', \tau_2', \Gamma_2', \tau_2''$ st $\Gamma_1 \vdash \mu_1 =_l \mu_2 \wedge \Gamma_2 \vdash \mu_1 =_l \mu_2 \wedge l|_{I_1} = l|_{I_2} \wedge \exists I_1', I_2', O_1, O_2. P_i, \mu_i(I_i) \rhd \langle I_i', O_i\rangle$ and one of the following holds:

$(1) P_1 = P_2 \wedge \Gamma_1 \vdash P_1 : L \wedge \Gamma_2 \vdash P_2 : L$
$(2) \langle I_1, P_1, \mu_1, \Gamma_1\rangle \mathcal{H}_l \langle I_2, P_2, \mu_2, \Gamma_2\rangle$
$(3) P_1 = P_1'; P' \wedge P_2 = P_2'; P' \wedge$
$\langle I_1, P_1', \mu_1, \Gamma_1\rangle \mathcal{H}_l \langle I_2, P_2', \mu_2, \Gamma_2\rangle \wedge P'$ typable by low

We show that $\mathcal{T}_l$ is a $\mathcal{B}_l$ bisimulation. We prove this by induction on $\mathcal{T}_l$.

Let $\langle I_1, P_1, \mu_1, \Gamma_1\rangle \mathcal{T}_l \langle I_2, P_2, \mu_2, \Gamma_2\rangle$ such that

$$\Gamma_1 \vdash \mu_1 =_l \mu_2 \wedge \Gamma_2 \vdash \mu_1 =_l \mu_2 \wedge l|_{I_1} = l|_{I_2} \wedge \exists I_1', I_2', O_1, O_2. P_i, \mu_i(I_i) \rhd \langle I_i', O_i\rangle$$

and $\Gamma_i, \tau_i \vdash P_i \ : \tau_i', (\Gamma_i', \tau_i'')$.

Case (1) We apply Lemma LOW to simulate the step and are left with

1) $P_1' = P_2'$ typable
2) $P_1' = P_1''; P' \wedge P_2' = P_2''; P' \wedge P_1' \neq P_2'$ and exists $\tau_1', \tau_2', \tau_1''', \tau_2'''$ such that $\Gamma_i', \tau_i'' \vdash P_i'' \ : \tau_i'', (\Gamma_i'', \tau_i''')$ and $\tau_i'' \not\sqsubseteq l$

We then conclude by inductive hypothesis on $\mathcal{T}_l$.

Case (2) We conclude by Proposition $\mathcal{H}_l$.

Case (3) We will analyze 4 cases.

Subcase $\langle P_1'; P', \mu_1\rangle \xrightarrow{*} \langle P', \mu_1'\rangle$ and $\langle P_2'; P', \mu_2\rangle \xrightarrow{*} \langle P', \mu_2'\rangle$.

We conclude by induction on $\mathcal{T}_l$ using the hypotheses of $\mathcal{H}_l$.

Subcase $\forall P''$ st $\langle P_1'; P', \mu_1\rangle \xrightarrow{*} \langle P'', \mu_1'\rangle$ then $P'' \neq P'$ and $\forall P''$ st $\langle P_2'; P', \mu_2\rangle \xrightarrow{*} \langle P'', \mu_2'\rangle$ then $P'' \neq P'$.

We conclude by $\mathcal{H}_l$.

Subcase $\langle P_1'; P', \mu_1\rangle \xrightarrow{*} \langle P', \mu_1'\rangle$ and $\forall P''$ st $\langle P_2'; P', \mu_2\rangle \xrightarrow{*} \langle P'', \mu_2'\rangle$ then $P'' \neq P'$.

We will show that this case is not possible. By hypothesis $\langle P_2'; P', \mu_2\rangle \xrightarrow{*} \langle P'', \mu_2'\rangle$, we know that P_2' must contain a while. Moreover, this while branches on a high variable since we have $\langle P_1'; P', \mu_1\rangle \xrightarrow{*} \langle P', \mu_1'\rangle$. Since P_2' is typed as high by hypothesis of $\mathcal{H}_l$then by rule of while, its effect is also high. Then by Lemma HIGH EFFECT P' should be typable as high, but this contradicts the hypothesis of case 3 of $\mathcal{T}_l$.

Subcase $\forall P''$ st $\langle P_1'; P', \mu_1\rangle \xrightarrow{*} \langle P'', \mu_1'\rangle$ then $P'' \neq P'$ and $\langle P_2'; P', \mu_2\rangle \xrightarrow{*} \langle P', \mu_2'\rangle$. Symmetric to previous case. □

Lemma XI.6. *Let* $I_1|_l = I_2|_l$ *and* $\Gamma \vdash \mu_1 =_l \mu_2$*. If* $\langle I_1, P, \mu_1, \Gamma\rangle \sim \langle I_2, P, \mu_2, \Gamma\rangle$ *then* P *is ITSNI.*

Proof. We will prove that: If $\langle I_1, P_1, \mu_1, \Gamma\rangle \sim \langle I_2, P_2, \mu_2, \Gamma\rangle$ then $I_1'|_l = I_2'|_l \wedge O_1|_l = O_2|_l$ where $P_1, \mu_1(I_1) \rhd \langle I_1', O_1\rangle$ and $P_2, \mu_2(I_2) \rhd \langle I_2', O_2\rangle$.

We will prove this by coinduction on the behaviour of two programs P_1, P_2.

If $I_1'|_l = I_2'|_l \wedge O_1|_l = O_2|_l$ where $P_1, \mu_1(I_1) \rhd \langle I_1', O_1\rangle$ and $P_2, \mu_2(I_2) \rhd \langle I_2', O_2\rangle$

Case $P1$ and P_2 are in the first case of the bisimulation and:

Subcase a_1 is an input event. Then by definition of behaviour $I_i'(ch) = a_i : I_i''(ch)$. We have to show $I_1''|_l = I_2''|_l$. This follows by definition of equality of streams and we conclude.

Subcase a_1 is an output event. Then by definition of behaviour $O_i'(ch) = a_i : O_i''(ch)$. We have to show $O_1''|_l = O_2''|_l$. This follows by definition of equality of streams and we conclude.

Case P_1 and P_2 are in the second case of the bisimulation:

Subcase P_1 is terminated and P_2 is not terminated. By second case of the bisimulation we have that P_2 makes step with an event not visible. If it is an input event, then by definition of behaviour $I_2'(ch) = a_2 : I_2''(ch)$ and we have that $I_1'|_l = []$. We have to prove that $I_1'|_l = I_2''|_l$. This follows because $I_1'|_l = a_2 : I_2''|_l$ by coinductive hypothesis and $\neq$ *visible*$_l(a_2)$ and we conclude.

Subcase P_2 is terminated and P_1 is not terminated. Symmetrical to the previous case.

Subcase P_2 and P_1 are not terminated. We apply a similar reasoning than the previous subcase to both P_1 and P_2.

Subcase P_2 and P_1 are both terminated. We conclude by hypothesis.

□

Theorem VII.1 (Soundness)**.** *If* $\Gamma \vdash P$ *then* P *is ITSNI.*

Proof. Direct by Lemma XI.5 and Lemma XI.6. □