A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor (Extended Version)

Andrew Bedford[a], Stephen Chong[c], Josée Desharnais[a], Elisavet Kozyri[b], Nadia Tawbi[a]

[a] *Université Laval, Québec, Canada*
[b] *Cornell University, New York, USA*
[c] *Harvard University, Massachusetts, USA*

Abstract

We present a novel progress-sensitive, flow-sensitive hybrid information-flow control monitor for an imperative interactive language. Progress-sensitive information-flow control is a strong information security guarantee which ensures that a program's progress (or lack of) does not leak information. Flow-sensitivity means that this strong security guarantee is enforced fairly precisely: our monitor tracks information flow per variable and per program point. We illustrate our approach on an imperative interactive language.

Our hybrid monitor is inlined: source programs are translated, by a type-based analysis, into a target language that supports dynamic security levels. A key benefit of this is that the resulting monitored program is amenable to standard optimization techniques such as partial evaluation.

One of the distinguishing features of our hybrid monitor is that it uses sets of levels to track the different *possible* security types of variables. This feature allows us to distinguish programs that *definitely* leak information from those that *may* leak information.

Keywords: information-flow control, non-interference, inline monitor, flow-sensitive, progress-sensitive

1. Introduction

We increasingly rely on computer systems to safeguard confidential information, and maintain the integrity of critical data and operations. But in our highly interconnected world, these trusted systems often need to communicate with untrusted parties. Trusted systems risk leaking confidential information to untrusted parties, or allowing input from untrusted parties to corrupt data or the operation of the trusted system.

Information-flow control is a promising approach to enable trusted systems to interact with untrusted parties, providing fine-grained application-specific control of confidential and untrusted information. Static mechanisms for information-flow control (such as security type systems [1, 2]) analyze a program before execution to determine whether the program's execution will satisfy the appropriate information flow requirements. This has low runtime overhead, but can generate many false positives. Dynamic mechanisms (e.g., [3]) accept or reject individual executions at runtime, without performing any static program analysis. Dynamic mechanisms can incur significant runtime overheads. More important, purely dynamic approaches cannot enforce information flow policies. These policies are not properties as defined by Schneider in [4], hence, one cannot decide whether or not the policy is satisfied by observing a single execution. *Hybrid information-flow control* techniques (e.g., [5]) combine static and dynamic program analysis and strive to achieve the benefits of both static and dynamic approaches: precise (i.e., per-execution) enforcement of security, with low runtime overhead.

In this work, we present a novel progress-sensitive [6], flow-sensitive hybrid information-flow control monitor for an imperative interactive language. It is an extension of the work presented at IFIP SEC 2016 [7]. The key features of our monitor are as follows.

Our monitor is progress-sensitive: it prevents confidential information from being leaked via *progress channels*. Information leaks through a progress channel when a program's progress (or lack of) depends on confidential information and is observable by an adversary. It is a generalization of termination-sensitive information security to interactive systems (i.e., systems that interact with an external environment at runtime).

Our monitor is *flow-sensitive*: the security level associated with program variables may change during the execution. Flow sensitivity increases precision (as the monitor is able to accept more programs) but complicates enforcement [8].

Our language has channel-valued variables: communication channels are constants that can be assigned to program variables. This language feature allows realistic communication scenarios to be modelled in our language (e.g., where the same code may communicate with users having arbitrary security levels), but complicates flow-sensitive enforcement of security. Most previous work on language-based information-flow control require that the channel used for an input or output operation be statically known, and allow only a single communication channel per security level.

Our monitor is inlined: source programs are translated into a target language that supports dynamic security levels [9]. The type-based translation inserts commands to track the security levels of program variables and contexts, and to control information flow. A key benefit of inlining the monitor is that the resulting monitored program in the target language is amenable to standard optimization techniques such as partial evaluation [10]. The instrumented code is thereby cleaned up so that the residual code tracks only the security levels of variables that cannot be determined statically.

Our monitor is hybrid: it uses both dynamic and static enforcement techniques. The translation to the target language performs a static analysis. If the program is statically determined to be insecure (i.e., it definitely contains a leak), then the program is statically rejected. Otherwise, the translation of the program dynamically tracks the statically unknown security levels of variables, and ensures that no leak occurs at runtime.

Our main contributions are as follows.

- We present an extended version of the hybrid monitor first presented in our previous article [7]. The extension consists in generalizing the flow- and progress-sensitive enforcement to general lattices. As in the conference paper, the combination of channel-valued variables, flow-sensitivity and progress-sensitivity presents a couple of issues that we have solved.

- The hybrid approach is based on the distinction between a leak of information that *may* occur from a leak that *definitely* occurs. This distinction is made possible using sets of security levels.

- We present two additional ideas to increase the precision of the static analysis and the permissiveness of the dynamic analysis: propagating constraints on the set of possible security levels and using conditional updates.

- We prove the soundness of our inlined monitor and that the semantics of the original program is preserved, as long as it is secure.

1.1. Examples

We present several examples of programs in our source language, to both provide background on information-flow control, and highlight some of the features of our hybrid monitor. For simplicity, we assume that the variables `lowValue`, `highValue`, `lowChannel`, and `highChannel` exist, have arbitrary values and have the suggested value security levels denoted L, for low, a public value, and H, for high, a private one.

Explicit and implicit flows. An *insecure explicit information flow* occurs when a confidential value is output to a public channel. An *insecure implicit information flow* [11] occurs when the decision to perform output on a public channel depends on confidential information. This violates security because an observer of the public channel will see whether the output occurred, and might thus learn confidential information. Techniques for tracking and controlling implicit and explicit information flow at the language level are well known [1, 2, 8], and are used in this work. The following program exhibits both explicit and implicit information flows, and our approach will reject this program statically.

```
(* insecure explicit flow *)
send highValue to lowChannel;
if highValue > 0 then
  (* insecure implicit flow *)
  send 42 to lowChannel
end
```

Unknown security levels. Our source language supports variables whose security level could be statically unknown. Consider the following program, where the output may or may not be secure, depending on the value of `lowValue`.

```
 if lowValue > 0
  then d := highChannel
  else d := lowChannel
 end;
 send highValue to d
```

Listing 1: Statically uncertain channel level

Purely static mechanisms would need to reject this program entirely, and indeed, to the best of our knowledge, all previous work either cannot express this program or reject it. This is because it would be unsound to statically treat `d` as being a private channel, since that might incorrectly allow private values to be sent to public channel `lowChannel`. Similarly, it would be unsound to statically treat `d` as a public channel, since that might incorrectly allow values read from private channel `highChannel` to be treated as public values. By contrast, our hybrid approach recognizes that the security of this program depends on the runtime value of `d`, and instruments it to track whether `d` refers to a high-security channel or a low-security channel, in order to intervene only in the latter case. We use sets of levels in our security types in order to track the different possibilities during our static analysis (e.g., `d` has type {L,H} after the conditional). To push the permissiveness a little more, we treat value variables similarly as channel variables.

Progress channels. The progress of a program, which is observable through its outputs, can also reveal information. For example, in the following program, whether or not the output on the low-security channel occurs depends on whether the preceding loop terminates, which in turn depends on confidential information.

```
while highValue > 0 do
  skip
end;
send 42 to lowChannel
```

Listing 2: Progress leak

Although this example leaks only 1 bit of information, progress channels can be used to leak a significant amount of information [6]. The most common way to prevent leaks through progress channels is to forbid loops whose execution depends on confidential information [12, 13], but it leads to the rejection of many secure programs, such as the following.

```
while highValue > 0 do
  highValue := highValue - 1
end; (*loop always terminates*)
send 42 to lowChannel
```

Listing 3: Loop that always terminates

To accept such programs, we follow Moore et al. [14] and use an oracle (conservative and assumed correct) to determine the termination behavior of loops. If the oracle determines that a loop always terminates (like the one in Listing 3), then we know that no following output could leak information through progress channels. On the other hand, if the oracle says that it may diverge, then we must take into account the fact that an output following the loop's execution could leak information.

In our approach, the oracle is a parameter and is based on termination analysis methods brought from the literature such as the one described in Cook et al. [15].

The inlined monitor injects another risk of leak through progress channels, as it can stop the execution. We treat this risk in the same way as divergence. We do not take into account leaks that can occur through timing channels.

Just as the one on channel variables, the permissiveness on value variables can result in a progress leak. This is illustrated in the following example, taken from Kozyri et al. [16].

```
if medValue > 0
   then x := highValue  (*x is H *)
   else x := lowValue   (*x is L *)
end;      (* x is {L,H} *)
"guarded send" x to unknownChannel
  (* executed only if there is no leak *)
send 1 to lowChannel
```

Listing 4: variable level sensitivity

If the unknown channel of the first **send** command happens to be of medium security level at runtime and if the monitor does not halt the execution, then the second **send** command would be reached. Then, if the output to `lowChannel` occurred, it would leak information about `medValue`. To prevent such a leak, we combine previous work [7] and the dynamic mechanism of Kozyri et al. [16].

1.2. Structure

In Section 2, we present the imperative language used to illustrate our approach. Section 3 defines the non-interference property. Section 4 describes our typed-based instrumentation mechanism, explains the type system, and presents the target language in which the instrumented programs are written; it is an extension of the source language with dynamic security levels. Section 5 proves that the instrumented programs are non-interferent. Section 6 presents two ways to increase the precision and permissiveness of our monitor. Section 7 is a summary of related work. Finally, we conclude in Section 8.

2. Source language

Source programs are written in a simple imperative language with commands for receiving and sending information.

We suppose that the interaction of a program with its environment (which can be a user or another program) is done through channels. Channels can be, for example, files, users, network channels, keyboards, computer screens, etc. The security levels of these channel constants are designated in advance by some security administrator. This is more realistic than requiring someone to manually define the level of every variable of the program; their level are instead given before the analysis and the execution as parameters. It also means that a program will not have to be rewritten if the privacy level of a channel changes over time: the analysis can be applied on the same program with the updated security levels for channels. The program has to be rewritten only if a flaw is detected.

2.1. Syntax

Let $\mathcal{V}$ be a set of identifiers for variables, and $\mathcal{C}$ a set of predefined communication channels. The syntax is as follows.

$$
\begin{array}{llll}
\textit{(variables)} & x & \in & \mathcal{V} \cup \mathcal{C} \\
\textit{(integer constants)} & n & \in & \mathbb{Z} \\
\textit{(expressions)} & e & ::= & x \mid n \mid e_1 \ \mathbf{op} \ e_2 \mid \mathbf{read} \ x \\
\textit{(commands)} & cmd & ::= & \\
\end{array}
$$
$$
\mathbf{skip} \mid x := e \mid \mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end} \mid \\
\mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end} \mid cmd_1; cmd_2 \mid \mathbf{send}\ x_1\ \mathbf{to}\ x_2
$$

Values are integers (we use zero for false and nonzero for true), or channel names. The symbol **op** stands for arithmetic or logic binary operators on integers. We write *Exp* for the set of expressions.

The core language is a standard imperative language. The expression **read** x returns the current value in channel x (without modifying it). Command **send** x_1 **to** x_2 sends the value of variable x_1 to channel x_2 and overwrites the current value in the channel. In other words, it outputs the value of x_1 to channel x_2. Without loss of generality, we consider that each channel consists of one value, it is easy to generalize to a channel consisting in a sequence of values.

$$(\textsc{Skip})\ \langle \mathbf{skip}, m, o\rangle \longrightarrow \langle \mathbf{stop}, m, o\rangle$$

$$(\textsc{Assign})\ \frac{m(e) = r}{\langle x := e, m, o\rangle \longrightarrow \langle \mathbf{stop}, m[x \mapsto r], o\rangle}$$

$$(\textsc{Send})\ \frac{m(x_1) = v \in \mathbb{Z} \qquad m(x_2) = ch \in \mathcal{C}}{\langle \mathbf{send}\ x_1\ \mathbf{to}\ x_2, m, o\rangle \longrightarrow \langle \mathbf{stop}, m[ch \mapsto v], o :: (v, ch)\rangle}$$

$$(\textsc{Seq1})\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle \mathbf{stop}, m', o'\rangle}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_2, m', o'\rangle}$$

$$(\textsc{Seq2})\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle cmd_1', m', o'\rangle \qquad cmd_1' \neq \mathbf{stop}}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_1'; cmd_2, m', o'\rangle}$$

$$(\textsc{If})\ \frac{m(e) \neq 0 \Longrightarrow i = 1 \qquad m(e) = 0 \Longrightarrow i = 2}{\langle \mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end}, m, o\rangle \longrightarrow \langle cmd_i, m, o\rangle}$$

$$(\textsc{Loop1})\ \frac{m(e) \neq 0}{\langle \mathbf{while}\, e\, \mathbf{do}\ cmd\, \mathbf{end}, m, o\rangle \longrightarrow \langle cmd; \mathbf{while}\, e\, \mathbf{do}\ cmd\, \mathbf{end}, m, o\rangle}$$

$$(\textsc{Loop2})\ \frac{m(e) = 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end}, m, o\rangle \longrightarrow \langle \mathbf{stop}, m, o\rangle}$$

Figure 1: Semantics of the source language

2.2. Semantics

A memory $m : \mathcal{V} \uplus \mathcal{C} \to \mathbb{Z} \uplus \mathcal{C}$ is a partial map from variables and channels to values, where the value of a channel is the last value sent to this channel. More precisely a memory is the disjoint union of two (partial) maps of the following form:

$$m_v : \mathcal{V} \to \mathbb{Z} \uplus \mathcal{C}, \qquad m_c : \mathcal{C} \to \mathbb{Z},$$

where $\uplus$ stands for the disjoint union operator. We omit the subscript whenever the context is clear. We write $m(e) = r$ to indicate that the evaluation of expression e under memory m returns r.

The semantics of the source language is mostly standard and is illustrated in Figure 1. Program configurations are tuples $\langle cmd, m, o\rangle$ where cmd is the command to be evaluated, m is the current memory and o is the current output trace. A transition between two configurations is denoted by the $\longrightarrow$ symbol. We write $\longrightarrow^*$ for the reflexive transitive closure of the $\longrightarrow$ relation.

We write $v :: vs$ for sequences where v is the first element of the sequence, and vs is the rest of the sequence. We write ϵ for the empty sequence. An output trace is a sequence of output events: it is of the form $o = (v_0, ch_0) :: (v_1, ch_1) :: \ldots$ where $v_k \in \mathbb{Z}$ is an integer value, and ch_k is a channel, $k \in \mathbb{N}$. The rule for sending a value appends a new output event to the end of the trace. We abuse notation and write $o :: (v, ch)$ to indicate event (v, ch) appended to trace o.

We write $\langle cmd, m, \epsilon\rangle \downarrow o$ if execution of configuration $\langle cmd, m, \epsilon\rangle$ can produce trace o, where o may be finite or infinite. For finite o, $\langle cmd, m, \epsilon\rangle \downarrow o$ holds if there is a configuration $\langle cmd', m', o\rangle$ such that $\langle cmd, m, \epsilon\rangle \longrightarrow^* \langle cmd', m', o\rangle$. For infinite o, $\langle cmd, m, \epsilon\rangle \downarrow o$ holds if for all traces o' such that o' is a finite prefix of o, we have $\langle cmd, m, \epsilon\rangle \downarrow o'$.

3. Security

For our purposes, we assume a finite lattice of security levels $(\mathcal{L}, \sqsubseteq)$ which contains at least two elements: L for the bottom of the lattice and H for the top of the lattice, i.e. $\forall \ell \in \mathcal{L}, L \sqsubseteq \ell \wedge \ell \sqsubseteq H$. We define an execution as ℓ-secure if the outputs to a channel of level ℓ do not reveal any information about the inputs of channels that are not lower than or equal to ℓ. This is a standard form of non-interference (e.g., [1, 2]) adapted for our particular language model.

Before formally defining non-interference, we first introduce some helpful technical concepts. The *projection of trace o to security level ℓ*, written $o \restriction \ell$, is its restriction to output events whose channels' security levels are less than or equal to ℓ. Formally,

$$\epsilon \restriction \ell = \epsilon$$
$$((v, ch) :: o) \restriction \ell = \begin{cases} (v, ch) :: (o \restriction \ell) & \text{if } \mathit{levelOfChan}(ch) \sqsubseteq \ell \\ o \restriction \ell & \text{otherwise} \end{cases}$$

where $\mathit{levelOfChan}(ch)$ denotes the security level of channel ch (typically specified by the administrator).

We say that two memories m and m' are ℓ-equivalent if they agree on the content of variables (including channel variables) whose security levels are ℓ or lower.

Definition 1 (Progress-sensitive non-interference) *We say that a program p satisfies* progress-sensitive non-interference *if for any $\ell \in \mathcal{L}$, and for any two memories m and m' that are ℓ-equivalent, and for any trace o such that $\langle p, m, \epsilon\rangle \downarrow o$, then there is some trace o', such that $\langle p, m', \epsilon\rangle \downarrow o'$ and $o \restriction \ell = o' \restriction \ell$.*

This definition of non-interference is progress-sensitive in that it assumes that an observer can distinguish an execution that will not produce any additional observable output (due to termination or divergence) from an execution that will make progress and produce additional observable output. Progress-insensitive definitions of non-interference typically weaken the requirement that $o \restriction \ell = o' \restriction \ell$ to instead require that $o \restriction \ell$ is a prefix of $o' \restriction \ell$, or vice versa.

Among the previously presented examples, only Listing 3 satisfies progress-sensitive non-interference. Nevertheless, we accept the program of Listing 1, since we transform it to a program that does satisfy progress-sensitive non-interference.

4. Type-based instrumentation

We enforce non-interference by translating source programs to a target language that enables the program to track the security levels of its variables. The translation performs a type-based static analysis of the source program, and rejects programs that definitely leak information (i.e., the translation fails).

In this section, we first present the security types for the source language (in order to provide intuition for the type-directed translation) followed by the description of the target language, which extends the source language with runtime representation of security levels. We then present the translation from the source language to the target language.

4.1. Source language types

Source language types are defined according to the following grammar. The security types are defined as follows:

(security labels, Lab)	$\ell ::= \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$
(value types, ValT)	$\sigma ::= int_\ell \mid int_\ell\, chan$
(variable types, VarT)	$\tau ::= \sigma_\ell$

Security labels are non-empty sets of security levels. They represent the possible security levels of a variable at runtime. If a security label contains more than one element, it means that its security level is statically unknown. For example, variable `x` of Listing 5 would be labeled $\{L, H\}$ after the **if** command as it could contain information of either level at runtime.

```
 if lowValue then
   x := read lowChannel
 else
   x := read highChannel
 end;
 send x to lowChannel
```

Listing 5: Variable of an uncertain level

We derive two order relations that allow us to deal with the sets of possible levels.

Definition 2 *The relations* $\sqsubseteq_s$, surely less than, *and* $\sqsubseteq_m$, maybe less than, *are defined as follows*

$$\begin{array}{ll} \ell_1 \sqsubseteq_s \ell_2 & \forall e_1 \in \ell_1, e_2 \in \ell_2 . e_1 \sqsubseteq e_2. \\ \ell_1 \sqsubseteq_m \ell_2 & \exists e_1 \in \ell_1, e_2 \in \ell_2 . e_1 \sqsubseteq e_2. \end{array}$$

Intuitively, we have $\ell \sqsubseteq_s \ell'$ when we can be sure statically that $\ell \sqsubseteq \ell'$ will be true at runtime, and we have $\ell \sqsubseteq_m \ell'$ when it is possible that $\ell \sqsubseteq \ell'$ at runtime.

Definition 3 *The supremum and infimum on sets of levels are defined as follows*

$$\ell_1 \sqcup \ell_2 = \{e_1 \in \ell_1, e_2 \in \ell_2 . e_1 \sqcup e_2\}$$
$$\ell_1 \sqcap \ell_2 = \{e_1 \in \ell_1, e_2 \in \ell_2 . e_1 \sqcap e_2\}$$

Value types are the types of integers and channels. Type int_ℓ is the type of integers whose values are of security level ℓ, and type $int_\ell\, chan$ is the type of a channel whose values are of security level ℓ.

Variables types associate a security level with a value type. Intuitively, $\sigma_{\ell'}$ represents the type of a variable whose value type is σ, and whose variable type is ℓ', the latter is an upper bound of the information level influencing the value of the variable. When a variable type ℓ is associated with a value type ℓ for a channel, it means that the sensitivity of the content of the channel is ℓ', and the sensitivity of the channel itself is ℓ. This approach was used by [7]. When a variable type ℓ is associated with a value type ℓ' for an integer, it means that the sensitivity of the integer is ℓ', and the sensitivity of ℓ' is ℓ'. This approach was used in [16] to prevent leakage through guarded sends.

Consider the program in Listing 1, page 2. Immediately following the conditional command, the type system gives variable `d` the type $(int_{\{L,H\}}\, chan)_{\{L\}}$. This type reflects that only low information determines which channel is assigned to `d` (i.e., variable `lowValue` determines `d`'s value), and whether `d` is a low channel or a high channel is statically unknown.

A similar situation occurs in Listing 4, page 3. Immediately following the conditional command, the type system gives variable `x` the type $(int_{\{L,H\}})_{\{M\}}$. This type reflects that an information of medium level determines which value is assigned to `x` (i.e., variable `medValue` determines `x`'s value), and that the information contained within `x` is either of low security level or of high security level. It is necessary to keep track of the context level in which `x` has been assigned its value. This information is used to halt the execution of the second **send** and prevent observers from deducing that `medValue` is less or equal to 0.

We instrument source programs to track at runtime the security levels that are statically unknown. In order to track these security levels, our target language allows their runtime representation.

4.2. Sets of levels

We use sets of levels not only to increase the precision of the analysis, but also because we *have to* due to our use of channel variables. Indeed, one of the issues that we encountered is the fact that we cannot conservatively approximate the level of a channel variable, due to Bell-Lapadula's principle of no write down, no read up. Listing 6 illustrates why.

```
if lowValue > 0 then
  d := lowChannel
else
  d := highChannel
end
send highValue to d (*Pessimist: d is L*)
x := read d (*Pessimist: d is H*)
send x to lowChannel
```

Listing 6: We cannot be pessimistic about channel variables

After the conditionals, `d` has type $(int_{\{L,H\}} chan)_{\{L\}}$ because it contains either a low or high channel and its value is assigned in a context of level L. Our typing system accepts this program, but makes sure that a runtime check is inserted. If the condition `lowValue > 0` happens to be true at runtime, then there is no leak. If it is false, the program will be rejected, thanks to the inserted runtime check controlling the send command. The uncertainty is unavoidable in the presence of flow sensitivity and channel variables. Indeed, we point out that we cannot be pessimistic about the level of `d` in this program. The output command suggests that a safe approximation for `d` would be a low security level. Yet, the input command suggests that a safe approximation for `d` would be a *high* security level, which contradicts the previous observation.

Consequently, in order to accept the program in Listing 6, we chose to use sets of security levels. As a consequence, we will obtain fewer false positives as we do not consider the worst possible case in our analysis, we leave it to the execution to check whether the information flow turns out to be secure or not.

4.3. Syntax and semantics of target language

Our target language is inspired by the work of Zheng and Myers [9], which introduced a language with first-class security levels, and a type system that soundly enforces non-interference in this language. The syntax of our target language is defined as follows. The main difference with the source language is that it adds support for *level variables* (regrouped in the set $\mathcal{V}_{level}$), a runtime representation of security levels.

(variables)	x	$\in$	$\mathcal{V} \cup \mathcal{C}$
(level variables)	$\tilde{x}$	$\in$	$\mathcal{V}_{level}$
(integer constants)	n	$\in$	$\mathbb{Z}$
(basic levels)	k	$\in$	$\mathcal{L}$
(level expressions)	ℓ	$::=$	$k \mid \tilde{x} \mid \ell_1 \sqcup \ell_2$
(integer expressions)	exp	$::=$	$x \mid n \mid exp_1 \ \mathbf{op} \ exp_2 \mid \mathbf{read}\ x$
(expressions)	e	$::=$	$exp \mid \ell$
(commands)	cmd	$::=$	

$$\begin{aligned}&\mathbf{skip} \mid (x_1, \dots, x_n) := (e_1, \dots, e_n) \mid \\ &\mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end} \mid cmd_1; cmd_2 \mid \\ &\mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end} \mid \mathbf{send}\ x_1\ \mathbf{to}\ x_2 \mid \\ &\mathbf{if}\ \ell_1 \sqsubseteq \ell_2\ \mathbf{then}\ (\mathbf{send}\ x_1\ \mathbf{to}\ x_2)\ \mathbf{else}\ \mathbf{fail}\ \mathbf{end}\end{aligned}$$

Dynamic types will allow a verification of types at runtime: this is the goal of the new send command, nested in a conditional – call it a *guarded send* – that permits to check some conditions on security levels before sending a given variable to a channel. If the check fails, the program aborts.

For simplicity, we assume that security levels can be stored only in a restricted set of variables $\mathcal{V}_{level} \subseteq \mathcal{V}$. Thus, the variable part m_v of a memory m now has the following type

$$m_v : (\mathcal{V}_{level} \to \mathcal{L}) \uplus (\mathcal{V} \setminus \mathcal{V}_{level} \to \mathbb{Z} \uplus \mathcal{C})$$

Furthermore we assume that $\mathcal{V}_{level}$ contains variables $\_\mathsf{pc}$ and $\_\mathsf{hc}$, and two level variables x_{val} and x_{ctx} associated with each variable $x \in \mathcal{V} \setminus \mathcal{V}_{level}$. They are used to track the security levels of variables at runtime. For example, if x is a channel variable of security type $(int_\ell\ chan)_{\ell'}$, then the values of these variables should be $x_{\mathsf{val}} = \ell$ and $x_{\mathsf{ctx}} = \ell'$ (this will be ensured by our instrumentation). Variables $\_\mathsf{pc}$ and $\_\mathsf{hc}$ hold the security levels of the context and halting context respectively, they represent the security level in which a command is executed.

The simultaneous assignment $(x_1, \dots, x_n) := (e_1, \dots, e_n)$ is introduced for the sake of clarity. Any assignment implies an immediate update of the concerned level variables. For all common commands, the semantics of the target language is the same as in the source language.

4.4. Instrumentation as a type system

Our instrumentation algorithm is specified as a type system in Figure 2. Its goal is to inline monitor actions in the program under analysis, thereby generating a safe version of it, or to reject it when it contains obvious leaks of information. The inlined actions are essentially updates of level variables and checks on these variables in order to control the execution of potentially leaking **send** commands. After a check, a **send** command is either executed if it is safe or its execution is prevented and the program is aborted.

The typing rules of variables and constants have judgements of the form $\Gamma \vdash e : \sigma_\ell$, stating that σ_ℓ is the type of e under the typing environment Γ. The instrumentation judgements are of the form

$$\Gamma, pc, hc \vdash cmd : t, h, \Gamma', [\![cmd]\!]$$

where $\Gamma, \Gamma' : \mathcal{V} \uplus \mathcal{C} \to VarT$ are typing environments (initially empty), cmd is the command under analysis, pc is the program context, hc is the halting context, t is the termination type of cmd, h is the updated halting context, and $[\![cmd]\!]$ is the instrumented command. The latter is often presented using a macro whose name starts with *gen*. The program context, pc, is used to keep track of the security level in which a command is executed, in order to detect implicit flows. The halting context, hc, is used to detect progress channels leaks. It represents the level of information that could have caused the program to *halt* (due to a failed guarded send command) or diverge (due to an infinite loop). In other words, it is the level of information that could be leaked through progress channels by an output. The termination t of a command is propagated in order to keep the halting context up to date. We distinguish three *termination types* $\mathcal{T} = \{T, D, M_\ell\}$, where T means that a command terminates for all memories, D, diverges for all memories, M_ℓ means that a command's termination is unknown statically; the subscript is used to indicate on which level(s) the termination depends. For example, the termination of the loop in Listing 2 is $M_{\{H\}}$ because it can either terminate or diverge at runtime, and

$$\text{(S-Chan)}\ \frac{\mathit{levelOfChan}(nch) = \ell}{\Gamma \vdash nch : (\mathit{int}_{\{\ell\}}\ \mathit{chan})_{\{L\}}} \qquad \text{(S-Int)}\ \Gamma \vdash n : (\mathit{int}_{\{L\}})_{\{L\}} \qquad \text{(S-Var)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(S-read)}\ \frac{\Gamma \vdash c : \mathit{int}_{\ell}\ \mathit{chan}_{\ell_c}}{\Gamma \vdash \textbf{read}\, c : (\mathit{int}_{\ell})_{\ell_c}}$$

$$\text{(S-Op)}\ \frac{\Gamma \vdash e_1 : (\mathit{int}_{\ell_1})_{\ell_1'} \qquad \Gamma \vdash e_2 : (\mathit{int}_{\ell_2})_{\ell_2'}}{\Gamma \vdash e_1\ \textbf{op}\ e_2 : (\mathit{int}_{\ell_1 \sqcup \ell_2})_{\ell_1' \sqcup \ell_2'}} \qquad \text{(S-Skip)}\ \Gamma, pc, hc \vdash\ \textbf{skip} :\ T, hc, \Gamma, \mathsf{skip}$$

$$\text{(S-Assign)}\ \frac{\Gamma \vdash e : \sigma_{\ell_e'}}{\Gamma, pc, hc \vdash x := e :\ T, hc, \Gamma[x \mapsto \sigma_{pc \sqcup \ell_e'}],\ \mathsf{genassign}}$$

$$\text{(S-Send)}\ \frac{\Gamma(x) = (\mathit{int}_{\ell_x})_{\ell_x'} \qquad \Gamma(c) = (\mathit{int}_{\ell_c}\ \mathit{chan})_{\ell_c'} \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell_x' \sqcup \ell_c') \sqsubseteq_s \ell_c}{\Gamma, pc, hc \vdash \textbf{send}\ x\ \textbf{to}\ c : T, hc, \Gamma, \textbf{send}\ x\ \textbf{to}\ c}$$

$$\text{(S-GSend)}\ \frac{\begin{array}{c}\Gamma(x) = (\mathit{int}_{\ell_x})_{\ell_x'} \\ \Gamma(c) = (\mathit{int}_{\ell_c}\ \mathit{chan})_{\ell_c'} \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell_x' \sqcup \ell_c') \not\sqsubseteq_s \ell_c \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell_x' \sqcup \ell_c') \sqsubseteq_m \ell_c\end{array}}{\Gamma, pc, hc \vdash \textbf{send}\ x\ \textbf{to}\ c : T, pc \sqcup hc \sqcup \ell_x' \sqcup \ell_c', \Gamma,\ \mathsf{gengsend}}$$

$$\text{(S-If)}\ \frac{\begin{array}{c}\Gamma \vdash e : (\mathit{int}_{\ell_e})_{\ell_e'} \qquad pc' = pc \sqcup \ell_e \sqcup \ell_e' \qquad h_d = \mathit{hasGSend}(\{\mathit{cmd}_1, \mathit{cmd}_2\}, pc') \\ h = \bigcup_{j \in \{1,2\}} (h_j \sqcup h_d \sqcup \mathit{level}(t_1 \oplus_{pc'} t_2)) \qquad \Gamma, pc', hc \vdash \mathit{cmd}_j : t_j, h_j, \Gamma_j, [\![\mathit{cmd}_j]\!] \quad j \in \{1,2\}\end{array}}{\Gamma, pc, hc \vdash\ \textbf{if}\ e\ \textbf{then}\ \mathit{cmd}_1\ \textbf{else}\ \mathit{cmd}_2\ \textbf{end} : t_1 \oplus_{pc'} t_2, h, \Gamma_1 \sqcup_{pc'} \Gamma_2, \mathsf{genif}}$$

$$\text{(S-Loop1)}\ \frac{\begin{array}{c}\Gamma = \Gamma \sqcup_{pc'} \Gamma' \qquad \Gamma \vdash e : (\mathit{int}_{\ell_e})_{\ell_e'} \qquad O(e, \mathit{cmd}, \Gamma) = t_o \qquad pc' = pc \cup (pc \sqcup \ell_e \sqcup \ell_e') \\ hc' = hc \cup (hc \sqcup \mathit{level}(t') \sqcup h') \qquad h_d = \mathit{hasGSend}(\{\mathit{cmd}\}, pc') \qquad \Gamma, pc', hc' \vdash \mathit{cmd} : t', h', \Gamma', [\![\mathit{cmd}]\!]\end{array}}{\Gamma, pc, hc \vdash \textbf{while}\ e\ \textbf{do}\ \mathit{cmd}\ \textbf{end} : t_o, h_d \sqcup h' \sqcup \mathit{level}(t_o), \Gamma \sqcup_{pc'} \Gamma', \mathsf{genwhile}}$$

$$\text{(S-Loop2)}\ \frac{\begin{array}{c}\Gamma \neq \Gamma \sqcup_{pc'} \Gamma' \qquad \Gamma \vdash e : (\mathit{int}_{\ell_e})_{\ell_e'} \qquad \Gamma, pc \sqcup \ell_e \sqcup \ell_e', hc \vdash \mathit{cmd} : t', h', \Gamma' \qquad pc' = pc \cup (pc \sqcup \ell_e \sqcup \ell_e') \\ hc' = hc \cup (hc \sqcup \mathit{level}(t') \sqcup h') \qquad \Gamma', pc', hc' \vdash \textbf{while}\ e\ \textbf{do}\ \mathit{cmd}\ \textbf{end} : t_o, h'', \Gamma'', \mathsf{instCode}\end{array}}{\Gamma, pc, hc \vdash \textbf{while}\ e\ \textbf{do}\ \mathit{cmd}\ \textbf{end} : t'', h'', \Gamma'', \mathsf{instCode}}$$

$$\text{(S-Seq1)}\ \frac{\Gamma, pc, hc \vdash \mathit{cmd}_1 : D, h, \Gamma_1, [\![\mathit{cmd}_1]\!]}{\Gamma, pc, hc \vdash \mathit{cmd}_1; \mathit{cmd}_2 : D, h, \Gamma_1, [\![\mathit{cmd}_1]\!]} \qquad \text{(S-Seq2)}\ \frac{\begin{array}{c}t_1 \neq D \qquad \Gamma, pc, hc \vdash \mathit{cmd}_1 : t_1, h_1, \Gamma_1, [\![\mathit{cmd}_1]\!] \\ \Gamma_1, pc, h_1 \vdash \mathit{cmd}_2 : t_2, h_2, \Gamma_2, [\![\mathit{cmd}_2]\!]\end{array}}{\Gamma, pc, hc \vdash \mathit{cmd}_1; \mathit{cmd}_2 : t_1 \mathbin{;} t_2, h_2, \Gamma_2, [\![\mathit{cmd}_1]\!]; [\![\mathit{cmd}_2]\!]}$$

Figure 2: Instrumentation and typing rules for the source language

this depends on information of level H. The loop in Listing 3 on the other hand is of termination type T because, no matter what the value of `h` is, it will always eventually terminate. Similarly, a loop whose condition is always true will have termination type D since it always diverges. Recall that, of course, the precision of this analysis depends on the precision of the oracle.

The instrumentation of a program p begins by inserting commands to initialize a few level variables: $\_\mathsf{pc}$, $\_\mathsf{hc}$ are initialized to L, as well as the level variables x_{ctx} and x_{val} for each variable $x \in \mathcal{V}$ appearing in p. Similarly, level variables c_{ctx} and c_{val} associated with each channel c used in p are also initialized, but the latter rather gets initialized to *levelOfChan*(c), which is an input parameter for the analysis. After the initialization, the instrumentation is given by the rules of Figure 2. We now explain these rules.

Rules (S-Chan) and (S-Int) specify the type of channel and integer constants respectively.

Rule (S-Var) infers the type of a variable from the environment Γ.

Rule (S-Op) infers the type of an expression from the types of its operands and it reflects the fact that operations on channels are not allowed.

Rule (S-Read) returns the security type of the current value in channel c.

Rule (S-Assign) updates the type of x in the environment as the supremum of the type of e (to prevent explicit flows) and the type of pc (to prevent implicit flows). Its instrumentation is given by the following macro, which represents a simultaneous assignment $x := e, x_{\mathsf{val}} := e_{\mathsf{val}}$, and $x_{\mathsf{ctx}} := e_{\mathsf{ctx}} \sqcup \_\mathsf{pc}$.

```
genassign =
  (x, x_val, x_ctx) := (e, e_val, e_ctx ⊔ _pc)
```

We write e_{ctx} to represent the expression made of the supremum of variables appearing in expression e. For example if $e = x + \mathbf{read}\ c$, then $e_{\mathsf{ctx}} = x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$. If $e = x + y$ then $e_{\mathsf{ctx}} = x_{\mathsf{ctx}} \sqcup y_{\mathsf{ctx}}$. The idea is the same for e_{val}.

Rule (S-Send) checks whether a send command is statically safe by requiring $(pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_s \ell_c$ (i.e., all possible values of the left-hand side are always lower or equal to the right-hand side). The variables on the left-hand side correspond to the level of information that can be revealed by the output to c. If so, the instrumentation inserts the send command as it is.

Rule (S-GSend) checks whether a send command may be safe, by requiring $(pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_m \ell_c$. The instrumentation then transforms it into a guarded send, as follows

```
gengsend =
    if $\_pc \sqcup \_hc \sqcup x_{val} \sqcup x_{ctx} \sqcup c_{ctx} \sqsubseteq c_{val}$ then
      (send $x$ to $c$)
    else
      fail
    end;
    $\_hc := \_pc \sqcup \_hc \sqcup x_{ctx} \sqcup c_{ctx}$;
```

The halting context must record the possible failure of a guarded send at runtime, and hence, it is updated with the level of information that influences its success/failure. Particularly, the halting context is updated with the sensitivity of the context and of the two variables involved, the channel variable [7] and the regular variable [16]. For example, Listing 7 shows why c_{ctx} must be included in this update.

```
if unknownValue > 0 (*H at runtime *)
  then c := lowChannel
  else c := highChannel
end;
send highValue to c (*guarded send*)
send lowValue to lowChannel
```

Listing 7: Dangerous runtime halting

Assume that `unknownValue` is high and false at runtime. Then the first guarded send is accepted, but allowing an output on a low security channel subsequently would leak information about `unknownValue`. This is because, had `unknownValue` been true, then the first send would have failed. However, `c` has level $int_{\{L,H\}}chan_{\{H\}}$ after the conditional. Updating $\_\mathsf{hc}$ with c_{ctx} will affect the check of all subsequent guarded send and prevent such leaks.

If none of the send rules can be applied, it means that the program under analysis contains a **send** command that always leaks information and so, the program is rejected.

Before explaining the rules for composed commands, we first define a few functions and operators. For the conditional rules, we need a supremum of environments.

Definition 4 *The supremum of two environments is given as* $dom(\Gamma_1 \sqcup_{pc} \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$, *and*

$$(\Gamma_1 \sqcup_{pc} \Gamma_2)(x) =$$
$$\begin{cases} \Gamma_i(x) & \text{if } x \in dom(\Gamma_i) \backslash dom(\Gamma_j), \\ & {}_{\{i,j\}=\{1,2\}} \vee \Gamma_1(x) = \Gamma_2(x) \\ (int_{\ell_1 \cup \ell_2}\, chan)_{(\ell'_1 \cup \ell'_2) \cup pc} & \text{if } \Gamma_1(x) = (int_{\ell_1}\ chan)_{\ell'_1} \\ & \wedge \Gamma_2(x) = (int_{\ell_2}\ chan)_{\ell'_2} \\ & \wedge \Gamma_1(x) \neq \Gamma_2(x) \\ (int_{\ell_1 \cup \ell_2})_{(\ell'_1 \cup \ell'_2) \cup pc} & \text{if } \Gamma_1(x) = (int_{\ell_1})_{\ell'_1} \wedge \\ & \Gamma_2(x) = (int_{\ell_2})_{\ell'_2} \wedge \\ & \Gamma_1(x) \neq \Gamma_2(x) \\ Error & \text{otherwise.} \end{cases}$$

When a typing inconsistency occurs, e.g., when a variable is used as an integer in one branch and as a channel in another, the analysis stops and an error is returned, causing the program to be rejected.

The function *level* : $\mathcal{T} \to \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$ returns the *termination level* (i.e., the level that the termination depends on) and is defined as:

$$level(t) = \begin{cases} \{L\} & \text{if } t \in \{T, D\} \\ \ell & \text{if } t = M_\ell \end{cases}$$

Two operators are used to compose terminations types, $\oplus$, used in the typing of conditionals, and $\fatsemi$, used in the typing of sequences. They are defined as follows.

$$t_1 \oplus_{pc} t_2 = \begin{cases} t_1 & \text{if } t_1 = t_2 \in \{T, D\} \\ M_{pc \sqcup (\ell_1 \cup \ell_2)} & \text{otherwise,} \\ & \ell_1 = \mathit{level}(t_1), \ell_2 = \mathit{level}(t_2) \end{cases}$$

$$t_1 \fatsemi t_2 = \begin{cases} M_{\ell_1 \sqcup \ell_2} & \text{if } t_1 = M_{\ell_1} \text{ and } t_2 = M_{\ell_2} \\ t_i & \text{if } t_j = T, \{i, j\} = \{1, 2\} \\ D & \text{otherwise} \end{cases}$$

The following example illustrates the use of these operators:

```
if highValue then
  while 1 do skip end (* D *)
else
  skip (* T *)
end
```

The termination type of an **if** command is computed using the $\oplus$ operator, from three parameters: the termination types of each of the two branches and the security level of the guard condition. Hence, in this example, we obtain $D \oplus_H T = M_{\{H\}}$.

One more function needs to be defined. Its motivation is given by the following example. Assume that in Listing 8 `unknownChannel` turns out to be a low channel at runtime. If the last send command is reached and executed it would leak information about `highValue`. The same leak would happen if instead of the guarded send we had a diverging loop.

```
if highValue > 0 then
 if ℓ ⊑ ℓ′
   then send highValue to unknownChannel
   else fail end;
end;
send lowValue to lowChannel
```

Listing 8: A guarded send can generate a progress leak

To prevent this kind of leak, we verify if the branches of a conditional contains guarded send commands using the function

$$\mathit{hasGSend} : \mathcal{P}(\mathit{Cmd}) \times \mathit{Lab} \to \mathit{Lab},$$

where Cmd is the set of commands and Lab the set of security labels. If one of the commands given in parameter contains a guarded send, then it returns the security label given in parameter, otherwise L is returned. This function is used to update the halting context to pc when there is a risk that one of the branch halts the execution. Without this update, hc could leak information about the condition in a subsequent send.

(S-If) When typing an **if** command, we type the two branches under pc', which is the supremum of the conditional's guard expression and current context. The resulting typing environments, Γ_1 and Γ_2, then contain the security levels that variables may have after executing the first or second branch. The typing environment returned by the **if** is the join of those two, defined in Definition 4, so that it contains the possibilities of both branches. Similarly, variable h is used to calculate the possible values that the halting context may have after the conditional, hence the union.

Its instrumentation is given by the following macro:

```
genif =   _oldpc^ν := _pc;               _pc := _pc ⊔ $e_{val}$ ⊔ $e_{ctx}$;
          if $e$ then                     ⟦$cmd_2$⟧;
          _pc := _pc ⊔ $e_{val}$ ⊔ $e_{ctx}$;   update($mv_1$)
          ⟦$cmd_1$⟧;                      end;
          update($mv_2$)                  _pc := _oldpc
          else
```

where update is defined as follows

```
update($mv$) =
      if $mv = \emptyset$ skip;
      else
        for each $x \in mv$   $x_{ctx} := x_{ctx} \sqcup$ _pc;
        if _hc $\in mv$   _hc := _hc ⊔ _pc;
```

and where t_j is the termination type of cmd_j, mv_j is the set of modified variables in cmd_j (we include _hc in this set if the termination of the two branches can differ i.e. if $\neg(t_1 = t_2 \in \{T, D\})$ or if at least one guarded send occurs in the other branch), and $e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$ is the guard condition's level expression.

The instrumented code starts by saving the current context to $\_\mathsf{oldpc}^\nu$ (the symbol ν indicates that it is a fresh variable). The program context is updated with the security level of the guard condition. The **if** itself is then generated.

The function update generates the command skip; if the parameter set mv is empty otherwise it generates updates of the context level of each modified variable in the other branch as well as the update of _hc if necessary. The underlying reason is to ensure that the value of these level variables is at least pc.

In a situation like the following listing, this function allows to update `x`'s level, to protect `unknownValue`.

```
x := 0;
if unknownValue then (*H at runtime*)
  x := 1
else skip end;
send x to lowChannel
```

Listing 9: Modified variables

Here, even if the else branch is taken at runtime, the level of variable `x` must be updated. Otherwise, information about `unknownValue` would be revealed by the send command (even with a guarded send).

(S-Loop1), (S-Loop2) rules specify the **while** command typing. They involve computing a fixed point to derive the right typing environment. This is necessary because of the flow sensitivity feature. Typing rule (S-Loop2) is applied recursively until a fixpoint is found, at which point (S-Loop1) is applied and its result is returned. The union operator is used to update the pc' and

hc' variables so that we keep track of all their possible values. Due to our use of finite lattices, and the monotonicity of the union and supremum on levels, it is easy to show that this computation converges, the proof is given in Appendix B, Lemma 7. The typing relies on O, an oracle that returns the termination type of the loop (t_o). It is worth noting that the call to the oracle is performed statically. Calling it dynamically would enhance precision, but increase significantly the overhead. If the loop contains guarded send commands, which could fail and reveal information about the condition of the loop, then we update the halting context to prevent this leak. The presence of at least one guarded send command is detected using the function $hasGSend()$.

```
genwhile = _oldpc^ν := _pc;                    end;
           while e do                          _pc := _pc ⊔ e_val ⊔ e_ctx;
           _pc := _pc ⊔ e_val ⊔ e_ctx;          update(mv);
           [[cmd]];                            _pc := _oldpc
```

The inserted commands are similar to those of the **if** command. The level variables and halting context are updated after the loop in case an execution does not enter the loop. The context needs to be updated at the begining of each iteration as the value, and hence level, of expression e may change.

(S-Seq1) is applied if cmd_1 always diverges; we then ignore cmd_2, as it will never be executed. Otherwise, (S-Seq2) is applied. The halting context returned is h_2 instead of $h_1 \sqcup h_2$ because h_2 already takes into account h_1.

Examples of instrumented programs are available in Appendix A.

5. Soundness

In order to prove that the instrumented program generated by Figure 2 correctly enforces non-interference, we need to adapt the definitions of non-interference and ℓ-equivalent memories to our target language, because of level variables. Recall that a memory for the target language is the union of two maps of the following form:

$$m_v : (\mathcal{V}_{level} \to \mathcal{L}) \uplus (\mathcal{V} \setminus \mathcal{V}_{level} \to \mathbb{Z} \uplus \mathcal{C}),$$
$$m_c : \mathcal{C} \to \mathbb{Z}.$$

We write $\mathrm{dom}_l(m) := \mathrm{dom}(m_v) \cap \mathcal{V}_{level} = m_v^{-1}(\mathcal{L})$. A memory m is called *complete* for a program p if

- $\{x_{\mathsf{val}}, x_{\mathsf{ctx}}\} \subseteq \mathrm{dom}_l(m)$ for any $x \in \mathcal{C} \cup \mathrm{dom}(m) \setminus \mathcal{V}_{level}$ that appears in p.
- _pc and _hc are in $\mathrm{dom}_l(m)$
- if $c \in \mathcal{C}$ appears in p, then $m_v(c_{\mathsf{val}}) = levelOfChan(c)$ and $m_v(c_{\mathsf{ctx}}) = L$
- if $m(x) \in \mathcal{C}$ then $m_v(x_{\mathsf{val}}) = levelOfChan(m(x))$.

The first two conditions ensure that level variables exist in the domain of the memory, whereas the last ones makes sure that it is compliant with the security policy for channels.

The definition of ℓ-equivalent memories, which is based on [16], must handle level variables. Whenever the level variable x_{ctx} corresponding to a variable x is such that $m(x_{\mathsf{ctx}}) \sqsubseteq \ell$ in one memory, then it must have the same value in both memories, otherwise a leak can happen.

Definition 5 *ℓ-equivalent memories. We say that two complete memories of the target language m_1 and m_2 are* ℓ-equivalent, *written $m_1 \equiv_\ell m_2$, iff they satisfy the following properties*

1. *if $m_i(\_\mathsf{pc}) \sqsubseteq \ell$ for some $i \in \{1,2\}$, then $m_1(\_\mathsf{pc}) = m_2(\_\mathsf{pc})$. The same property holds for* _hc.
2. *$x \in m_{i,v}^{-1}(\mathbb{Z} \cup \mathcal{C}), \wedge m_i(x_{\mathsf{ctx}}) \sqsubseteq \ell$, for some $i = 1,2$, then*
   - $m_1(x_{\mathsf{ctx}}) = m_2(x_{\mathsf{ctx}})$
   - $m_1(x_{\mathsf{val}}) = m_2(x_{\mathsf{val}})$
   - *if $m_1(x_{\mathsf{val}}) \sqsubseteq \ell$ then $m_1(x) = m_2(x)$*
3. *$c \in \mathcal{C} \wedge levelOfChan(c) \sqsubseteq \ell \Rightarrow m_{1_c}(c) = m_{2_c}(c)$*

It may seem surprinsing that the memories may differ on a level variable such as x_{val}; this is because they may differ on the value of high variables. Too see this, here is a variation of Listing 1.

```
if highValue > 0
 then d := highChannel
 else d := lowChannel
end;
```

Listing 10: The security type of `d`'s content is sensible

In this example, the content of `d` depends on a private condition, and hence its level variable $\mathtt{d}_{\mathsf{ctx}}$ should be H; moreover, variable $\mathtt{d}_{\mathsf{val}}$, containing the level of the content of `d`, may have different values in two ℓ-equivalent memories. Another example, for non-channel variables, is Listing 4, reproduced below (and taken from Kozyri et al. [16]), where `medValue` has security level M, with $L \sqsubseteq M \sqsubseteq H$.

```
if medValue > 0
 then x := highValue  (*x is H *)
 else x := lowValue   (*x is L *)
end;      (* x is {L,H} *)
send x to unknownChannel
send 1 to lowChannel
```

In this example, variable x_{val} contains information of level M. If `unknownChannel` turns out to be of level M at execution then reaching or not the last send command will depend on x_{val}. The program will be blocked or not, and this will reveal which branch was taken in the preceding conditional. Interestingly, this problem does not arise when the lattice is restricted to only two levels, $\{L, H\}$, as argued in Kozyri et al.

Here is the definition of non-interference for the target language. The difference from Definition 1 is the requirement of the memories to be complete and the use of Definition 5.

Definition 6 *Progress-sensitive non-interference. We say that a program p satisfies* progress-sensitive non-interference *if for any $\ell \in \mathcal{L}$, and for any two complete memories m and m' that are ℓ-equivalent, and for any trace o such that $\langle p, m, \epsilon\rangle \downarrow o$, then there is some trace o', such that $\langle p, m', \epsilon\rangle \downarrow o'$ and $o \lceil \ell = o' \lceil \ell$.*

Using these updated definitions, we prove that the instrumented programs are non interferent.

Theorem 1 (Soundness of enforcement) *If a program p is well typed according to the type system of Figure 2, then the generated program $[\![p]\!]$ satisfies progress sensitive non-interference.*

We also show that the instrumentation preserves the semantics of the original program. That is, the instrumentation of a program p, written $[\![p]\!]$, produces exactly the same output as p as long as it is allowed to continue; it may be stopped at some point to prevent a leak. If m is a memory for the target language we write $\hat{m}$ for the restriction of m to $\mathcal{V} \setminus \mathcal{V}_{level}$.

Theorem 2 (Semantics preservation) *Let p be a program, m a memory, and o, o' output traces. Then*

$$\langle [\![p]\!], m, \epsilon\rangle \downarrow o \Rightarrow \ \langle p, \hat{m}, \epsilon\rangle \downarrow o$$
$$(\langle p, \hat{m}, \epsilon\rangle \downarrow o \wedge \langle [\![p]\!], m, \epsilon\rangle \downarrow o') \Rightarrow \ o \preceq o' \vee o' \preceq o$$

where $o' \preceq o$ means that o' is a prefix of o.

The proofs are available in Appendix B.

6. Increasing precision and permissiveness

During the course of this work, we thought of two ways to improve the precision of our static analysis and permissiveness of our dynamic analysis. While we chose not to use them into this work (to keep things as simple as possible), we think they are worth pointing out.

6.1. Security type constraints

In Listing 12, only executions where `c` is a high channel will get past the first guarded send. For this reason, we can consider, for the rest of the analysis, that its type is $int_{\{H\}} chan_{\{L\}}$ instead of $int_{\{L,H\}} chan_{\{L\}}$.

```
if lowValue > 0 then
  c:= lowChannel
else
  c:= highChannel
end;
send highValue to c; (*will be transformed
   into a guarded send*)
(*to reach here, c must be {H}*)
x := read c; (*so x is {H}*)
send x to lowChannel (*always leaks, so
  statically rejected*)
```

Listing 12: Constraint on the security type of a channel variable

The same idea applies to integer variables. For example, we know that the instructions after the first **send** of Listing 13 will only be reached if variable `x` is low. For this reason, we can consider that `x`'s type after the **send** will be $(int_L)_{\{L\}}$ instead of $(int_{\{L,H\}})_{\{L\}}$.

```
if lowValue > 0 then
  c := lowChannel
else
  c := highChannel
end;
x := read c; (*x is {L,H}*)
send x to lowChannel; (*will be
  transformed into a guarded send*)
(*to reach here, x must be {L}*)
send x to lowChannel (*no need to
  transform into a guarded send*)
```

Listing 13: Constraint on the security type of an integer variable

Using these constraints in our static analysis would increase the detection rate of programs that definitely leak information, rather than having them fail at runtime.

6.2. Conditional updates of the halting context

While we chose to always update the halting context `_hc` after a guarded send with

$$\_\mathsf{hc} := \_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}},$$

there are cases where we can be more precise. One such case is illustrated in Listing 14.

```
if medValue then
  x := read highChannel
else
  x := read highChannel2
end;
send x to unknownChannel; (*guarded send*)
send lowValue to lowChannel
```

Listing 14: Example where `_hc` does not need to be updated with $\mathsf{x}_{\mathsf{ctx}}$

In this example, while `x`'s value may differ, its type is constant and equal to $(int_{\{H\}})_{\{M\}}$. Since the value of `medValue` does not affect its type, it means that it has no influence on the guarded send's decision to block or allow the output. Hence, in this case, the update to `_hc` variable after the guarded send does not need to include variable $\mathsf{x}_{\mathsf{ctx}}$.

Similarly, when a channel `c` whose type is constant is used in a guarded send command, the update to `_hc` does not need to include variable $\mathsf{c}_{\mathsf{ctx}}$.

Hence, using conditional updates would allow the dynamic analysis to be more permissive as the updates to `_hc` are less conservative.

7. Related work

There has been much research into language-based techniques for controlling information flow over the last two decades. In this section, we focus on hybrid techniques for information-flow control.

Hybrid techniques are attractive as they have the potential to offer the advantages of both static and dynamic analyses: the low runtime overhead of static approaches combined with the precision and flexibility of dynamic techniques.

Le Guernic et al. [5] present the first hybrid information-flow control monitor. The enforcement is based on a monitor that is able to perform static checks during the execution. The enforcement is not flow-sensitive. Le Guernic, in [17], extends this work to concurrent programs. Russo and Sabelfeld [8] generalize their work, presenting a series of hybrid monitors that differ on the action to perform in the event of a security violation. They also state that purely dynamic enforcements are more permissive than purely static enforcements but they cannot be used in case of flow-sensitivity. They propose a hybrid flow-sensitive enforcement based on calling static analysis during the execution. This enforcement is not progress sensitive.

Kozyri et al. [16] show that it is not trivial to design dynamic enforcement mechanisms that support general lattices and do not leak information through termination. In particular, they show that labels on labels are necessary for lattices with more than two elements, but that two levels of labels is enough.

Bedford et al. [18] generate instrumented code, enforcing information flow based on static analysis (i.e., an information-flow monitor is inlined). The approach supports channel variables and is flow sensitive, but does not take into account leaks due to progress. Also, the inlined monitor does not use dynamic security levels, but employs a heavy-handed approach which is not as amenable to standard optimization techniques as the present one. The target language is not formally defined and no soundness proof of the instrumented code is provided.

Moore et al. [14] consider precise enforcement of flow-insensitive progress-sensitive security. Progress sensitivity is also based on an oracle's analysis, but they call upon it dynamically while we do it statically. We have also introduced additional termination types to increase the permissiveness of the monitor.

Chudnov and Naumann [19] inline a flow-sensitive hybrid monitor (based on a monitor of Russo and Sabelfeld [8]) and extend it to Javascript [20]. They prove its soundness by showing that the execution of the inlined monitor is bisimilar to the execution of a non-inlined monitor. We inline a flow-sensitive progress-sensitive hybrid monitor and, as we did not have already have a non-inlined monitor, we proved its soundness by showing that the output traces produced by two ℓ-equivalent executions will always be the same. Magazinius et al. [21] present on-the-fly inlining of a dynamic information security monitor. We speculate that we could extend their ideas to allow on–the fly instrumentation.

Askarov and Sabelfeld [22] use hybrid monitors to enforce information security in dynamic languages. In this setting, dynamic evaluation of programs (e.g., eval statements in JavaScript) requires on-the-fly static analysis of programs. They provide a model to define non-interference that is suitable to progress-sensitivity and they quantify information leaks due to termination [6].

Hritcu et al. [23] introduces an error-handling mechanism that allows all errors (even those caused by an information-flow control violation) to be safely recoverable. They support dynamic levels. To help prevent leaks through covert channels, they provide a discretionary access control mechanism called clearance that allows them to put an upper bound on the *pc*. Contrarily to our approach, the detection (and prevention) of leaks through progress channels is not done automatically.

Askarov et al. [24] introduce a hybrid monitoring framework capable of handling concurrent programs. They illustrate their approach on a simple imperative language similar to ours, but it does not support channel variables. In their framework, each thread is guarded by its own local monitor (progress- and flow-sensitive). There is also a single global monitor that synchronizes the threads. Like us, they make use of an oracle to approximate the termination behaviour of branches. This oracle is called upon at runtime (making it a kind of on-the-fly static analysis), whereas ours is called only statically. The main difference between their approach and ours, exluding the concurrency of course, is the fact that our monitor is inlined whereas theirs is not.

8. Conclusion

We have presented a hybrid information flow enforcement mechanism, which detects and prevents leaks that may occur through the data-flow or the progress of a program. It uses information inferred during a phase of static analysis to instrument the program; this helps to reduce the number of false positives during the execution. The instrumented program uses level variables, a simple yet powerful way, to perform its dynamic analysis. This instrumented code can then be partially evaluated in order to reduce the amount of added commands.

Our main contributions are the following.

(a) We present an extended version of the hybrid monitor first presented in our previous article [7]. It is capable of enforcing flow- and progress-sensitive information security on general lattices. It is more precise and introduces less overhead than currently available solutions (e.g., [12, 14]) for two reasons: it makes use of a static termination oracle and does not approximate the level of a variable at the join of a conditional. Since our monitor is inlined, it can be easily optimized using classical partial evaluation techniques, [10].

(b) We prove the soundness of our inlined monitor and that the semantics of the original program is preserved, as long as it is secure.

(c) We show that, thanks to the use of sets of levels, it is possible to distinguish programs that *definitely* contain leaks of information from programs that *may* leak information.

(d) We present two ideas to increase the precision of the static analysis and the permissiveness of the dynamic analysis: propagating constraints on the set of possible security levels and using conditional updates. We chose not to use them in this work in order to increase readability.

Future work includes extensions to concurrency, declassification and information leakage due to timing. We would like to scale up the approach to deal with real world languages and to test it on elaborate programs. The use of abstract interpretation, [28], to enhance the static analysis is also to be considered in future work.

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

[2] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of computer security*, vol. 4, no. 2, pp. 167–187, 1996.

[3] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2009.

[4] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[5] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," *Asian Computing Science Conference*, 2006.

[6] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proceedings of the European Symp. on Research in Computer Security: Computer Security*, 2008.

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

[8] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *CSF*. IEEE Computer Society, 2010, pp. 186–199.

[9] L. Zheng and A. C. Myers, "Dynamic security labels and noninterference," in *Formal Aspects in Security and Trust*. Springer, 2005, pp. 27–40.

[10] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

[11] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976.

[12] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *CSFW*. IEEE, 2006.

[13] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL*, 1998.

[14] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *CCS 2012*, 2012.

[15] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, no. 5, pp. 88–98, May 2011.

[16] E. Kozyri, J. Desharnais, and N. Tawbi, "Block-safe information flow control," Department of Computer Science, Cornell University, Tech. Rep., Aug. 2016.

[17] G. L. Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *CSF*, 2007.

[18] A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi, "Enforcing information flow by combining static and dynamic analysis," in *Proceedings of the International Symposium on Foundations & Practice of Security*, 2013.

[19] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 23rd IEEE Security Foundations Symposium*, 2010.

[20] ——, "Inlined information flow monitoring for javascript," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 629–643. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813684

[21] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," *Computers & Security*, vol. 31, no. 7, pp. 827–843, 2012.

[22] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *CSF*, 2009.

[23] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your ifcexception are belong to us," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 3–17.

[24] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *Computer Security Foundations Symposium*, 2015.

[25] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a java-like language." in *CSFW*, vol. 2, June 2002, p. 253.

[26] D. Hedin and D. Sands, "Noninterference in the presence of non-opaque pointers," in *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*, July 2006, pp. 217–229.

[27] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2006, pp. 91–102.

[28] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252.

Appendix A. Examples

In order to simplify the examples, we assume that the variables `lVal`, `mVal`, `hVal` and channels `lChan`, `mChan`, `hChan` already exist, have random values and have the suggested security levels. We also assume that all other level variables have been initialized to L. Boolean values are represented as integer where 0 means false and any other integer means true.

Guarded send

The following example illustrates a situation where the guarded send is used to prevent a possible leak of information.

```
if  mVal > 0  then
  c := mChan
else
  c := hChan
end;
send hVal to c; (*may leak information*)
```

Here is its instrumentation :

```
(*if*)
$\_oldpc_1 := \_pc$;
if $lVal > 0$ then
  $\_pc := \_pc \sqcup lVal_{val} \sqcup lVal_{ctx} \sqcup L$;

  (*assign*)
  $(c,\ c_{val},\ c_{ctx}) := (lChan,\ lChan_{val},\ lChan_{ctx} \sqcup \_pc)$;

  $(c, c_{ctx}) := (c, c_{ctx} \sqcup \_pc)$;  (*update*)
else
  $\_pc := \_pc \sqcup lVal_{val} \sqcup lVal_{ctx} \sqcup L$;

  (*assign*)
  $(c,\ c_{val},\ c_{ctx}) := (hChan,\ hChan_{val},\ hChan_{ctx} \sqcup \_pc)$;

  $(c, c_{ctx}) := (c, c_{ctx} \sqcup \_pc)$;  (*update*)
end
$\_pc := \_oldpc_1$;

(*guarded send*)
if $\_hc \sqcup \_pc \sqcup hVal_{val} \sqcup hVal_{ctx} \sqcup c_{ctx} \sqsubseteq c_{val}$ then
  (send hVal to c)
else fail end;
$\_hc := \_pc \sqcup \_hc \sqcup hVal_{ctx} \sqcup c_{ctx}$;
```

Divergence

Commands after a loop that always diverges are ignored. Hence, the following program is statically safe.

```
while 1 do
  skip
end;
send hVal to lChan
```

Even if it is statically safe, it is instrumented :

```
(*while*)
$\_oldpc_1 := \_pc$;
while 1 do
  $\_pc := \_pc \sqcup L \sqcup L$;
  (*skip*)
  skip;
end;
$\_pc := \_pc \sqcup L \sqcup L$;
$\_pc := \_oldpc_1$
```

After partial evaluation, it results in the following program:

```
while 1 do
  skip
end
```

Notice that because the code is statically safe, the partial evaluation is able to get rid of the instructions added by the instrumentation algorithm. This is because, if the code is statically safe, then the conditions of the guarded send commands are all true. If guarded send commands are not needed, then level variables are also not needed.

Appendix B. Proofs

We prove that the type system of Figure 2 generates non-interferent programs (i.e., we prove its soundness).

Theorem 1 (Soundness of enforcement) *If a program p is well typed according to the type system of Figure 2, then the generated program $[\![p]\!]$ satisfies progress sensitive non-interference.*

The theorem is a consequence of the following results.

Lemma 3 *If two memories are ℓ-equivalent, then they agree on every expressions involving level variables that were generated by the instrumentation and whose value is $\sqsubseteq \ell$; these expressions include $\_pc, \_hc, e_{val}, e_{ctx}$, etc.*

Sketch. The proof is by induction. The idea is to show that all such expressions either only use level variables of the form x_{ctx} which are $\sqsubseteq \ell$, on which ℓ-equivalent memories agree by definition, or that whenever these expressions use some x_{val}, then they *protect* this potentially dangerous variable by x_{ctx}. The program context, $\_pc$, is modified in the conditional and the loop, where the supremum is taken from the level of the condition. $\square$

Notation If $\Gamma(x) = \sigma_{\ell'_x}$, we write $\Gamma_{ctx}(x)$ to mean ℓ'_x. If $\Gamma(x) = (int_{\ell_x}\ chan)_{\ell'_x}$ or $(int_{\ell_x})_{\ell'_x}$, then we write $\Gamma_{val}(x)$ to mean ℓ_x.

The following lemma states that security types of variables calculated by the typing system of Figure 2 include all possible runtime values.

Lemma 4 *For any execution starting with a complete memory, if the runtime memory is m when an instrumented command $\llbracket cmd \rrbracket$ is executed, and m' is the memory after the execution, assuming the typing generated by Figure 2 is*

$$\Gamma, pc, hc \vdash cmd : t, hc', \Gamma', \llbracket cmd \rrbracket,$$

we have

(1) $m(\_\mathsf{pc}) \in pc$ *and* $m(\_\mathsf{pc}) = m'(\_\mathsf{pc})$

and the following invariants in the program execution

(2) $m(\_\mathsf{hc}) \in hc$

(3) $m(x_{\mathsf{val}}) \in \Gamma_{\mathsf{val}}(x)$

(4) $m(x_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(x)$

Proof. The proof is by structural induction.
Base Cases:
Initially, we have $pc = hc = \{L\}$, $m(\_\mathsf{pc}) = m(\_\mathsf{hc}) = L$, similarly for x_{val} and x_{ctx}, so the conditions are true at the initial state.

Case $\llbracket$**skip**$\rrbracket$ is trivial.

Case $\llbracket x := e \rrbracket = (x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) := (e, e_{\mathsf{val}}, e_{\mathsf{ctx}} \sqcup \_\mathsf{pc})$
Since the execution of this code does not modify $\_\mathsf{pc}$ or $\_\mathsf{hc}$, we have (1) and (2).

For every variable t_{val} in expression e_{val}, we have that $m(t_{\mathsf{val}}) \in \Gamma_{\mathsf{val}}(t)$. Similarly, for every variable t_{ctx} in expression e_{ctx}, we have that $m(t_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(t)$. Hence, by the definition of the supremum on sets (Definition 3) and (S-Op), we have (3) (i.e., $m'(x_{\mathsf{val}}) \in \Gamma'_{\mathsf{val}}(x)$). We also have (4) due to the updated environment returned (S-Assign).

Case $\llbracket$**send** x **to** $c\rrbracket$
Subcase $\llbracket$**send** x **to** $c\rrbracket$ = **send** x **to** c
Trivial since no variable type is modified.

Subcase $\llbracket$**send** x **to** $c\rrbracket$ =

```
if $\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}} \sqsubseteq c_{\mathsf{val}}$
then (send $x$ to $c$)
else fail
end;
$\_\mathsf{hc} := \_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$;
```

Initially we have that $m(\_\mathsf{pc}) \in pc$, $m(\_\mathsf{hc}) \in hc$, $m(x_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(x)$ and $m(c_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(c)$. Only $\_\mathsf{hc}$ is modified by this command, hence we have (1), (3) and (4). We also have (2) by the supremum on sets and the hc' returned by (S-GSend).

Induction Cases:
Case $\llbracket cmd_1; cmd_2 \rrbracket$
We can use induction on $\llbracket cmd_1 \rrbracket$, with the following hypothesis:

- $\Gamma, pc, hc \vdash cmd_1 : t_1, hc_1, \Gamma_1, \llbracket cmd_1 \rrbracket$.

Induction hypothesis gives us that m'' (the memory after executing $\llbracket cmd_1 \rrbracket$) and Γ_1 satisfy (1)-(4). If $t_1 = D$ then $\llbracket cmd_1; cmd_2 \rrbracket$ is simply $\llbracket cmd_1 \rrbracket$, and we are done since all executions diverge. If $t_1 \neq D$, then we have for $\llbracket cmd_2 \rrbracket$ that:

- $\Gamma_1, pc, h_1 \vdash cmd_2 : t_2, h_2, \Gamma_2, \llbracket cmd_2 \rrbracket$

Since the same program context as for cmd_1 is fed to the typing rule, then by the induction hypothesis and the fact that m'' and Γ_1 satisfy (1)-(4), we have the result for m' and Γ_2.

Case $\llbracket$**if** e **then** cmd_1 **else** cmd_2 **end**$\rrbracket$ =

```
$\_\mathsf{oldpc}^{\nu}$ := $\_\mathsf{pc}$;
if $e$ then
$\_\mathsf{pc} := \_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
$\llbracket cmd_1 \rrbracket$;
update($mv_2$)
else
$\_\mathsf{pc} := \_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
$\llbracket cmd_2 \rrbracket$;
update($mv_1$)
end;
$\_\mathsf{pc} := \_\mathsf{oldpc}$
```

For (1) we observe that $\_\mathsf{pc}$ has the same value before and after the instrumented code. Moreover, before cmd_i, $i = 1, 2$, the value of $\_\mathsf{pc}$ is in pc', the context used to type these commands. For (3), by induction, we have that the invariants are true after the execution of $\llbracket cmd_1 \rrbracket$ or $\llbracket cmd_2 \rrbracket$. Hence we have (3). The update function in each branch inserts commands that, for every modified variable x, updates its x_{ctx} with $\_\mathsf{pc}$. This corresponds to what is done in the supremum of environments (Definition 4), so we have (4). Finally, since the update function updates the $\_\mathsf{hc}$ with $\_\mathsf{pc}$ when the termination behavior of the two branches can differ, and that the $\_\mathsf{hc}$ is updated by update if one of the branch contains a guarded send, we have (2) as this corresponds to the union done in (S-If).

Case $\llbracket$**while** e **do** cmd **end**$\rrbracket$

```
$\_\mathsf{oldpc}^{\nu} := \_\mathsf{pc}$;
while $e$ do
$\_\mathsf{pc} := \_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
$\llbracket cmd \rrbracket$;
end;
$\_\mathsf{pc} := \_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
update($mv$);
$\_\mathsf{pc} := \_\mathsf{oldpc}$
```

The case is similar to the conditional. Variable $\_\mathsf{pc}$ is updated at the beginning of each iteration, it belongs to $pc \sqcup \ell_e \sqcup \ell'_e$, as wanted for the induction step involving $\llbracket cmd \rrbracket$, and we get (1), as pc' contains all the possible values of $\_\mathsf{pc}$ when $\llbracket cmd \rrbracket$ is executed. By induction, we have that the invariants are true after executing $\llbracket cmd \rrbracket$. Hence we have (3). We also have (2) and (4) using the same arguments as in the conditional.

□

The following proposition shows that any step of two executions performed from ℓ-equivalent memories results in ℓ-equivalent outputs. Theorem 1 follows as a corollary from it and from the next lemma.

Proposition 5 *Let m_i, $i = 1, 2$ be two ℓ-equivalent complete memories, o_i be a trace and $[\![cmd]\!]$ be a command generated by Figure 2, that is*

$$\Gamma, pc, hc \vdash cmd : t, h, \Gamma', [\![cmd]\!].$$

Then if both $[\![cmd]\!], m_i$ terminate, that is, we have maximal executions $\langle [\![cmd]\!], m_i, o_i\rangle \longrightarrow^* \langle \mathbf{stop}, m_i', o_i'\rangle$, then the following statements are invariants:

(a) the memories are ℓ-equivalent (Def 5)

(b) ℓ-projections of observations are equal

Proof. The proof is by structural induction.
Base Cases:
Initially, we have m_i, $i = 1, 2$ two ℓ-equivalent memories where the level variables have been initialized, so (a) and (b) are straightforward.

Case $[\![\mathbf{skip}]\!]$ is trivial.

Case $[\![x := e]\!] = (x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) := (e, e_{\mathsf{val}}, e_{\mathsf{ctx}} \sqcup \_\mathsf{pc})$
We have to prove (a), as (b) is trivial since this command does not modify the output traces. For ℓ-equivalence, since _pc, _hc and the contents of channels are not modified, we obtain conditions (1) and (3) of Definition 5. For condition (2): since m_1 and m_2 are ℓ-equivalent, if $m_1'(x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) \neq m_2'(x, x_{\mathsf{val}}, x_{\mathsf{ctx}})$, then by Lemma 3 it implies that $m_i'(x_{\mathsf{ctx}}) = m_i(\_\mathsf{pc} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$. Meaning that there is at least one variable t_{ctx} in e_{ctx} such that $m_i(t_{\mathsf{ctx}}) \not\sqsubseteq \ell$, or that $\_\mathsf{pc} \not\sqsubseteq \ell$. Hence, m_1' and m_2' are still ℓ-equivalent. If $m_1'(x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) = m_2'(x, x_{\mathsf{val}}, x_{\mathsf{ctx}})$, then m_1' and m_2' are ℓ-equivalent.

Case $[\![\mathbf{send}\ x\ \mathbf{to}\ c]\!]$
Subcase $[\![\mathbf{send}\ x\ \mathbf{to}\ c]\!] = \mathbf{send}\ x\ \mathbf{to}\ c$
By the typing rule, we have

$$pc \sqcup hc \sqcup \ell_x \sqcup \ell_x' \sqcup \ell_c' \sqsubseteq_s \ell_c$$

using the property of supremum, combining with Lemma 4 and converting the notation, we obtain

$$\begin{aligned} m_i(\_\mathsf{pc}) &\in pc \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(\_\mathsf{hc}) &\in hc \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(x_{\mathsf{val}}) &\in \Gamma_{\mathsf{val}}(x) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(x_{\mathsf{ctx}}) &\in \Gamma_{\mathsf{ctx}}(x) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(c_{\mathsf{ctx}}) &\in \Gamma_{\mathsf{ctx}}(c) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \end{aligned}$$

By the definition of $\sqsubseteq_s$, this implies $m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq m_i(c_{\mathsf{val}})$, which means that the sending of x on c is safe. Hence we obtain (a) and (b) by ℓ-equivalence of the memories.

Subcase $[\![\mathbf{send}\ x\ \mathbf{to}\ c]\!] =$

```
  if  $\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}} \sqsubseteq c_{\mathsf{val}}$
    then (send $x$ to $c$)
    else fail
 end;
 $\_\mathsf{hc} := \_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$;
```

We have to prove (a) and (b). For (a), we only have to prove conditions (1) and (3) as only the value of _hc and the content of the channel may change.

If $m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq \ell$ for one memory $i \in \{1, 2\}$ then, by ℓ-equivalence, it is also the case for the other memory and we get (1). It also implies that the two memory agree on x_{val} and c_{val} as they are ℓ-equivalent. This means that they either both fail, in which case there is nothing more to prove, or the **send** command is executed in both. If it is executed, then $m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq m_i(c_{\mathsf{val}})$, for $i = 1, 2$, and by ℓ-equivalence again, $o_i \upharpoonright \ell = o_i' \upharpoonright \ell$ in both executions and (b) is true, as well as (3).

If $m_i'(\_\mathsf{hc}) = m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \not\sqsubseteq \ell$ for $i = 1, 2$, then we get (1) and we have that the **send** command could succeed in m_1 and fail in m_2. But since $m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \not\sqsubseteq \ell$, we know that it cannot succeed on a channel whose level is $\sqsubseteq \ell$ and we get (b) and (3).

Induction Cases:
Case $[\![cmd_1; cmd_2]\!]$
The hypothesis gives that m_1 and m_2 satisfy (a) and (b). Then we can use induction on $[\![cmd_1]\!]$, with the following hypotheses:

- $\Gamma, pc, hc \vdash cmd_1 : t_1, h_1, \Gamma_1, [\![cmd_1]\!]$.
- $\langle [\![cmd_1]\!], m_i, o_i\rangle \longrightarrow^* \langle \mathbf{stop}, m_i'', o_i''\rangle$

Induction hypothesis gives us that m_1'' and m_2'' satisfy (a) and (b). We have for $[\![cmd_2]\!]$ that:

- $\Gamma_1, pc, h_1 \vdash cmd_2 : t_2, h_2, \Gamma_2, [\![cmd_2]\!]$
- $\langle [\![cmd_2]\!], m_i'', o_i''\rangle \longrightarrow^* \langle \mathbf{stop}, m_i', o_i'\rangle$

By the induction hypothesis and the fact that m_1'' and m_2'' satisfy (a) and (b), we obtain that m_1' and m_2' satisfy (a) and (b).

Case $[\![\mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end}]\!] =$

```
_oldpc$^\nu$  := _pc;
if $e$ then
_pc  := $\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
$[\![cmd_1]\!]$;
update($mv_2$)
else
_pc  := $\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$;
$[\![cmd_2]\!]$;
update($mv_1$)
end;
_pc := _oldpc
```

Assume that m_1 and m_2 satisfy (a) and that $o_1 \upharpoonright \ell = o_2 \upharpoonright \ell$. If $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) = \sqcup_{t \in Var(e)} m_i(t_{\mathsf{val}} \sqcup t_{\mathsf{ctx}}) \sqsubseteq \ell$, $i = 1, 2$ (symmetry is given by ℓ-equivalence), we have the result by induction since both memories take the same branch, say i.

Now assume that $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, $i = 1, 2$. If $m_1(e) = m_2(e)$ we are in the same situation as above. So assume w.l.o.g. that $m_1(e)$ is true but $m_2(e)$ is false; hence under memory m_i, code $[\![cmd_i]\!]$ will be executed.

If both commands terminate, that is:

- $\langle [\![cmd_1]\!], m_1, o_1\rangle \longrightarrow^* \langle \mathbf{stop}, m'_1, o'_1\rangle$
- $\langle [\![cmd_2]\!], m_2, o_2\rangle \longrightarrow^* \langle \mathbf{stop}, m'_2, o'_2\rangle$

Then we know, by Lemma 6(c), that

$$o'_1 \upharpoonright \ell = o_1 \upharpoonright \ell = o_2 \upharpoonright \ell = o'_2 \upharpoonright \ell.$$

because before executing cmd_i, variable $\_\mathsf{pc}$ is updated with the value of $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}})$ which is $\not\sqsubseteq \ell$. Hence, the execution of $[\![cmd_i]\!]$ will produce no output on channels of level $\sqsubseteq \ell$. Hence, (b) is proven for the induction step.

This also proves (a) on channels, but we need to prove it on variables (level and non level).

Let $x \in \mathcal{V}$ and $mv = mv_1 \cup mv_2$, where mv_i the set of variables that may be modified in cmd_i, $i = 1, 2$. By definition, variable $\_\mathsf{hc}$ is also included in this set if the termination of the branches may differ or if one of them contains a guarded send. If $x \notin mv$ and the m_i's agree on x, x_{ctx} and x_{val}, then the m'_i's also agree on x, x_{ctx} and x_{val}.

If $x \in mv_i$, we have that $x_{\mathsf{ctx}} \sqsupseteq \_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$ due to the commands inserted by the update function. Thus we have (a).

Case $[\![\mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end}]\!] =$

```
_oldpc^ν := _pc;                 end;
while e do                       _pc := _pc ⊔ e_val ⊔ e_ctx;
_pc := _pc ⊔ e_val ⊔ e_ctx;      update(mv);
[[cmd]];                         _pc := _oldpc
```

As a loop is essentially a (possibly infinite) sequence of **if**, the case is similar to the conditional. Assume that m_1 and m_2 satisfy (a) and (b). If $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) = \sqcup_{t \in Var(e)} m_i(t_{\mathsf{val}} \sqcup t_{\mathsf{ctx}}) \not\sqsubseteq \ell$, then we know that the execution of $[\![cmd]\!]$ will produce no output on channels of level $\sqsubseteq \ell$ since variable $\_\mathsf{pc}$ is updated before entering the loop, and that every variable that is or could have been modified by $[\![cmd]\!]$ will have a t_{ctx} that is $\not\sqsubseteq \ell$ due to the update function. Hence, we have (a) and (b).

If instead we have that $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$, then we have the result by induction since both memories will always take the same branch.

□

Lemma 6 *With the premises of the previous theorem, if one step of $[\![cmd]\!]$ fails or diverges for one memory then it also fails or diverges for the other or no more output on channels of level ℓ or lower will be performed on that execution. More precisely, for $i = \{1, 2\}$*

(a) $m_i(\_\mathsf{hc}) \sqsubseteq m'_i(\_\mathsf{hc})$

(b) if only one execution fails or diverges, say m_1, then $o_2 \upharpoonright \ell = o'_2 \upharpoonright \ell$ and $m'_2(\_\mathsf{hc}) \not\sqsubseteq \ell$.

(c) if $m_i(\_\mathsf{pc}) \not\sqsubseteq \ell$ or $m_i(\_\mathsf{hc}) \not\sqsubseteq \ell$ then $o_i \upharpoonright \ell = o'_i \upharpoonright \ell$

(d) if both executions fail or both diverge, then the ℓ-projections of observations are equal

Proof. (a) is straightforward since $\_\mathsf{hc}$ is always included in the right-hand side when updating $\_\mathsf{hc}$.

(b) is proven by induction, following the lines of Proposition 5. The interesting cases are the guarded send subcase and the inductive cases.

Case $[\![\mathbf{send}\ x\ \mathbf{to}\ c]\!]$
Let's assume that the send command is transformed into a guarded send. The only case where only one execution fails, say m_1, is one where $m'_2(\_\mathsf{hc}) \not\sqsubseteq \ell$, and where no observation is made on a channel of level ℓ or lower, as wanted.

Case $[\![cmd_1; cmd_2]\!]$
Let m''_i and o''_i be the memories and output traces after executing cmd_1. Let m'_i and o'_i be the memories and output traces after executing $cmd_1; cmd_2$. If one execution fails or diverges, say m_1 on cmd_1, then by induction, $m''_2(\_\mathsf{hc}) \not\sqsubseteq \ell$, and by (c), we obtain $o_2 \upharpoonright \ell = o''_2 \upharpoonright \ell$. By (a) and (c) and induction, we then get $o_2 \upharpoonright \ell = o'_2 \upharpoonright \ell$, as wanted. By (a), and (c) again, we also obtain $m'_2(\_\mathsf{hc}) \not\sqsubseteq \ell$. If m_1 fails or diverges on cmd_2 instead, the argument is similar.

Case $[\![\mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end}]\!]$
There are two situations in which only one execution, say m_1, can fail: (1) only one execution executes a guarded send or (2) they both execute the same guarded send but the result is different. In case (1), we have that $m_i(\_\mathsf{pc}) \not\sqsubseteq \ell$ (a consequence of Proposition 5 and Lemma 3), and that at least one of the **if**'s branch contains a guarded send, which is always followed by an update to variable $\_\mathsf{hc}$. Since one of the branch modifies variable $\_\mathsf{hc}$, it also means that the update function updates $\_\mathsf{hc}$ to $\_\mathsf{pc}$, which is $\not\sqsubseteq \ell$. Hence, we have (b) in this case. In case (2), the only way that the result of a guarded send can be different is if $m_i(\_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup \mathsf{x}_{\mathsf{ctx}} \sqcup \mathsf{c}_{\mathsf{ctx}}) \not\sqsubseteq \ell$, where x is the variable sent and c the channel on which the send occurs. In this case, the update to $\_\mathsf{hc}$ that immediately follows the guarded send will ensure that $m_2(\_\mathsf{hc}) \not\sqsubseteq \ell$. Hence we also have (b) in this case.
Similarly, for only only one execution to diverge, two things must be true: (1) $m_i(\_\mathsf{pc}) \not\sqsubseteq \ell$, and (2) the termination type of the **if** command is M. Since the termination type is M, we have that the update function inserted after the conditional updates the $\_\mathsf{hc}$ to $\_\mathsf{pc}$, which is $\not\sqsubseteq \ell$. Hence, we also have (b) in this case.

Case $[\![\textbf{while } e \textbf{ do } cmd \textbf{ end}]\!]$
The argument is similar to the **if** command.

For (c), there are two cases where there are observations, the send case and the guarded send case. For the latter, the condition is taken care of by the guard. For the former, if $m_i(\_\mathsf{pc}) \not\sqsubseteq \ell$ or $m_i(\_\mathsf{hc}) \not\sqsubseteq \ell$ then, by Lemma 4 (1) and (2), one of the sets pc or hc contains a security level $\ell' \not\sqsubseteq \ell$. By the typing rule, this implies that $\ell' \sqsubseteq_s \Gamma_{\mathsf{val}}(c)$. By definition of $\sqsubseteq_s$, all elements of $\Gamma_{\mathsf{val}}(c)$ are greater than or equal to ℓ' and hence, again by Lemma 4 (3), $\ell' \sqsubseteq_s m_i(c_{\mathsf{val}})$ and $\ell \not\sqsubseteq_s m_i(c_{\mathsf{val}})$, thus $o_i \restriction \ell = o'_i \restriction \ell$.

Finally, for (d), we have three cases: both executions fail, both executions diverge and $m_i(\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$ or both executions diverge and $m_i(\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, where e is the guard expression of the loop that diverges. If both executions fail, then there will be no more outputs and so the ℓ-projections remain equivalent. If both executions diverge and $m_i(\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$, then by Lemma 3 we have that the ℓ-projections of observations are equal. If both executions diverge and $m_i(\_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, then the update to the $\_\mathsf{pc}$ inside the body of the loop ensures that $m_i(\_\mathsf{pc}) \not\sqsubseteq \ell$ before executing cmd, and so that there will be no more outputs on channels of level lower or equal to ℓ.

□

Lemma 7 (Fixed-point) *The fixed-point computed by typing rules* (S-Loop1) *and* (S-Loop2) *always converges to a value.*

Proof. First let's observe that the pc' computed by S-Loop always reaches a fixed-point due to the fact that it is a set of levels on which we add elements at each iteration; elements are never removed. Hence, in the worst case, pc' will be equal to $\mathcal{L}$ (since $\mathcal{L}$ is a finite set).

Let $x \in \mathrm{dom}(\Gamma)$.

We have that $\Gamma_{\mathsf{val}}(x) \subseteq (\Gamma \sqcup_{pc'} \Gamma')_{\mathsf{val}}(x)$ by Definition 4, as it states that $(\Gamma \sqcup_{pc'} \Gamma')_{\mathsf{val}}(x) = \Gamma_{\mathsf{val}}(x) \cup \Gamma'_{\mathsf{val}}(x)$. Hence, $\Gamma_{\mathsf{val}}(x)$ ultimately reaches a fixed point since $\mathcal{L}$ is finite.

Depending on the type associated to pc', we either have that $\Gamma_{\mathsf{ctx}}(x) \subseteq (\Gamma \sqcup_{pc'} \Gamma')_{\mathsf{ctx}}(x)$ or that the elements of $\Gamma_{\mathsf{ctx}}(x)$ are replaced by levels that are greater than them, by Definition 4 and Definition 3. Since pc' reaches a fixed point, the subsequent modifications of $\Gamma_{\mathsf{ctx}}(x)$ amounts to a union operation. Hence, the typing of $\Gamma_{\mathsf{ctx}}(x)$ reaches a fixed point eventually as the lattice $\mathcal{L}$ is finite.

□

Theorem 2 *(Semantics preservation.) Let p be a program, m a memory, and o an output trace. Then*

$$\langle [\![p]\!], m, \epsilon\rangle \downarrow o \Rightarrow \ \langle p, \hat{m}, \epsilon\rangle \downarrow o$$
$$(\langle p, \hat{m}, \epsilon\rangle \downarrow o \wedge \langle [\![p]\!], m, \epsilon\rangle \downarrow o') \Rightarrow \ o \preceq o' \vee o' \preceq o$$

where $o' \preceq o$ means o' is a prefix of o.

Note that $\hat{m}$ is a memory for the source language, and hence it is of type : $\mathcal{V} \uplus \mathcal{C} \to \mathbb{Z} \uplus \mathcal{C}$ whereas m can, in addition, map variables to levels.

Sketch. By structural induction. The program generated by our instrumentation contains the same commands as the original program, in the same order. The only difference being the additional assignments on level variables and checks. For this reason, the only non-trivial case is the send command, since it modifies the output trace, or halts the program. Hence assume that $cmd = \textbf{send } x \textbf{ to } c$. There are two cases: (1) $[\![cmd]\!] = cmd$ and (2) $[\![cmd]\!] =$ $\textbf{if } \_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}} \sqsubseteq_s c_{\mathsf{val}} \textbf{ then } (\textbf{send } x_1 \textbf{ to } x_2)$ $\textbf{else fail end}; \_\mathsf{hc} := \_\mathsf{pc} \sqcup \_\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$.

In the first case, the claim is trivial. In the second case, the send is guarded by a condition. If this condition is true, the sending will happen and the output trace will be updated with $o{::}(m(x_1), m(x_2))$, as would have been done by cmd. Otherwise, the program $[\![cmd]\!]$ will be stopped, and hence, no more output will happen, although cmd could produce other outputs. □