# A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor

Andrew Bedford[1] (✉), Stephen Chong[2], Josée Desharnais[1], and Nadia Tawbi[1]

[1] Laval University, Quebec, Canada
andrew.bedford.1@ulaval.ca
[2] Harvard University, Cambridge, 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: we track information flow according to the source of information and not to an a priori given variable security level. 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.

## 1 Introduction

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 [12,14]) analyse a program before execution to determine whether its execution satisfies the information flow requirements. This has low runtime overhead, but can generate many false positives. Dynamic mechanisms (e.g., [4]) accept or reject individual executions at runtime and thus can incur significant runtime overheads. *Hybrid information-flow control* techniques (e.g., [8]) combine static and dynamic program analysis and strive to achieve the benefits of both: precise (i.e., per-execution) enforcement of security and low runtime overhead.

We present a novel progress-sensitive [2], flow-sensitive hybrid information-flow control monitor for an imperative interactive language. Our monitor prevents leaks of confidential information, notably via *progress channels*, while limiting over approximation, thanks to *flow sensitivity* and its inline nature. Our monitor is inlined: source programs are translated into a target language that supports dynamic security levels [15]. The type-based translation inserts commands to track the security levels of program variables and contexts, and to

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
J.-H. Hoepman and S. Katzenbeisser (Eds.): SEC 2016, IFIP AICT 471, pp. 352–366, 2016.
DOI: 10.1007/978-3-319-33630-5_24

control information flow. A key benefit is that the resulting monitored program is amenable to standard optimization techniques such as partial evaluation [7].

The translation to the target language performs a static analysis using three security levels: $L$ (for low-security information), $H$ (for high-security information), and $U$ (for unknown information). If the program is statically determined to be insecure, then it is rejected. Otherwise, the translation of the program dynamically tracks the unknown security levels, and ensures that no leak occurs.

Our main contributions are twofold. This work is one of the first hybrid monitor that enforces both flow and progress-sensitive information security; moreover, the combination of channel-valued variables, flow-sensitivity and progress-sensitivity presents a couple of issues that we solve.

## Motivating Examples

*Channel Variables.* Our source language supports channel variables whose security level can be statically unknown. This leads to use a special security level, $U$, which delays the decision to accept or reject certain programs to runtime. Indeed, a channel level needs upward or downward approximation according to its use and this cannot be approximated, as the following example shows.

```
    if lowValue  > 0
      then  d := lowChannel
      else  d := highChannel end;
(* Case 1  *)                    (* Case 2 *)
send highValue to d               x := read d;
(*to be rejected if d is L*)    send x to lowChannel
                                (*to be rejected if d is H*)
```

Listing 1.1: We cannot be pessimistic about channel variables

*Progress Channels.* The progress of a program, observable through its outputs, can reveal information. In the following program, the occurrence of an output on the public channel reveals a confidential information controlling the loop termination.

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

Listing 1.2: Progress leak

The most common way to prevent leaks through progress channels is to forbid loops whose execution depends on confidential information [10,13], but it leads to the rejection of many secure programs, such as the following.

```
while highValue > 0 do
  highValue := highValue - 1 end;
send 42 to lowChannel
```

Listing 1.3: Loop that always terminates

Inspired by Moore et al. [9], we use an oracle to determine the termination behaviour of loops. If it tells that a loop always terminates (cf Listing 1.3), then there is no possible leak of information. If the oracle says it may diverge, then a risk of information leak is flagged. The oracle is a parameter based on termination analysis methods brought from the literature [6].

**Structure.** In Sect. 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 is a summary of related work. Finally, we conclude in Sect. 6.

## 2 Source Language

Source programs are written in a simple imperative language. We suppose that the interaction of a program with its environment is done through channels. Channels can be, for example, files, users, network channels, keyboards, etc. These channel constants are associated to a priori security levels, private or public. This is more realistic than requiring someone to manually define the level of every variable of the program; their level can instead be inferred according to the sources of information they may hold.

### 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.

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

Values are integers (we use zero for false and nonzero for true), or channel names. Symbol **op** stands for arithmetic or logic binary operators. We write *Exp* for the set of expressions. W.l.o.g., we assume each channel consists of one value, which can be read or modified through **read** operation and **send** command respectively. It is easy to generalize to channels consisting in sequences of values.

### 2.2 Semantics

A memory $m : \mathcal{V} \uplus \mathcal{C} \rightarrow \mathbb{Z} \uplus \mathcal{C}$ 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 maps of the following form:

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

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 Fig. 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 $\epsilon$ 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 $o$.)

$$\text{(Skip)}\quad \langle \textbf{skip}, m, o\rangle \longrightarrow \langle \textbf{stop}, m, o\rangle \qquad \text{(Assign)}\ \frac{m(e) = r}{\langle x := e, m, o\rangle \longrightarrow \langle \textbf{stop}, m[x \mapsto r], o\rangle}$$

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

$$\text{(Seq1)}\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle \textbf{stop}, m', o'\rangle}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_2, m', o'\rangle} \qquad \text{(Seq2)}\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle cmd_1', m', o'\rangle \qquad cmd_1' \neq \textbf{stop}}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_1'; cmd_2, m', o'\rangle}$$

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

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

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

**Fig. 1.** Semantics of the source language

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

We define an execution as secure if the outputs on public channels do not reveal any information about the inputs of private channels. This is a standard form of non-interference (e.g., [12,14]) adapted to our particular language model. More formally, we require that any two executions of the programs starting from initial memories that have the same public channel inputs, produce the same publicly observable outputs. This means that an observer of the public output could not distinguish the two executions, and thus learns nothing about the inputs of private channels.

Before formally defining non-interference, we first introduce some helpful technical concepts. We assume a lattice of security levels $(\mathcal{L}, \sqsubseteq)$ with two elements: $L$ (Low) for public information and $H$ (High) for private information, ordered as $L \sqsubseteq H$. The *projection of trace $o$ to security level $\ell$*, written $o \restriction \ell$, is its restriction to output events whose channels' security levels are less than or equal to $\ell$. 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 $m'$ *differ only on private channel inputs* if $m_v = m'_v$ and

$$\forall ch \in \mathcal{C}. \mathit{levelOfChan}(ch) = L \Rightarrow m_c(ch) = m'_c(ch).$$

**Definition 1 (Progress-Sensitive Non-Interference).**
*We say that a program $p$ satisfies* progress-sensitive non-interference *if for any two memories $m$ and $m'$ that agree on public variables and public channel inputs, and for any (finite or infinite) 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 $o \restriction L = o' \restriction L$.*

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 $o \restriction L = o' \restriction L$ to instead require that $o \restriction L$ is a prefix of $o' \restriction L$, or vice versa.

## 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 clearly 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:

$$\begin{array}{ll} \textit{(security levels, } \mathcal{L}\textit{)} & \ell ::= \ L \mid U \mid H \\ \textit{(value types, ValT)} & \sigma ::= \textit{int} \mid \textit{int}_\ell \ \textit{chan} \\ \textit{(variable types, VarT)} & \tau ::= \sigma_\ell \end{array}$$

Security levels in types include $L$ and $H$, and also $U$ (Unknown), which is used to represent a statically unknown security level. The translated program will explicitly track these statically unknown security levels at runtime. The security levels are organized in a lattice $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L} = \{L, U, H\}$ and $L \sqsubseteq U \sqsubseteq H$, $(H \not\sqsubseteq U \not\sqsubseteq L)$. The associated supremum is denoted $\sqcup$. We derive two order relations that allow us to deal with the uncertainty level.

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

$$\begin{array}{ll} \ell_1 \sqsubseteq_s \ell_2 & \textit{if } (\ell_1 \sqsubseteq \ell_2) \wedge \neg(\ell_1 = \ell_2 = U) \\ \ell_1 \sqsubseteq_m \ell_2 & \textit{if } (\ell_1 \sqsubseteq \ell_2 \vee \ell_1 = U \vee \ell_2 = U) \end{array}$$

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. For example, $U \not\sqsubseteq_s L$ but $U \sqsubseteq_m L$.

Value types are the types of integers (*int*) and channels. Type $\textit{int}_\ell \ \textit{chan}$ is the type of a channel whose values are of security level $\ell$.

Variables types associate a security level with a value type. Intuitively, $\sigma_\ell$ represents the type of a variable whose value type is $\sigma$, and whose variable type is $\ell$, the latter is an upper bound of the information level influencing the value.

We instrument source programs to track at runtime the security levels that are statically unknown. That is, if a variable $x$ has type $\sigma_U$ for some value type $\sigma$, then the instrumented program will have a variable that explicitly tracks the security level of variable $x$. Moreover, if $\sigma$ is the unknown channel type ($\textit{int}_U \ \textit{chan}$) then the instrumented program will have a variable that explicitly tracks the security level of the channel that is assigned to $x$. In order to track these security levels, our target language allows their runtime representation.

**The Uncertain Level.** As illustrated in Listing 1.1, a channel level needs upward or downward approximation according to its use. This is the main reason underlying the use of the uncertainty level $U$. After the conditionals of that listing, `d` has type $(\textit{int}_U \textit{chan})_L$ 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 in both `Case 1` and `Case 2`, but inserts runtime checks. If the condition `lowValue` $>$ `0` is false at runtime, then sending of a `highValue` on `d` would be safe, and `Case 1` should be accepted, while `Case 2` should be rejected since it attempts to send a high level value to a public channel. On the contrary, if `lowValue` $>$ `0` appears to be false at runtime, then `Case 1` should be accepted and `Case 2` rejected.

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 variable channels in this program. The output command suggests that a safe (yet too strong) approximation for `d` would be a low security level. Yet, the input command suggests that a safe (yet too strong) approximation for `d` would be a *high* security level, which contradicts the previous observation. Consequently, if we are to accept the program in Listing 1.1, in both cases, we need an alternative security type, $U$, to carry on with the analysis.

## 4.2 Syntax and Semantics of Target Language

Our target language is inspired by the work of Zheng and Myers [15], 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.

$$
\begin{array}{lrcl}
\textit{(variables)} & x & \in & \mathcal{V} \cup \mathcal{C} \\
\textit{(level variables)} & \tilde{x} & \in & \mathcal{V}_{level} \\
\textit{(integer constants)} & n & \in & \mathbb{Z} \\
\textit{(basic levels)} & k & ::= & L \mid H \\
\textit{(level expressions)} & \ell & ::= & k \mid \tilde{x} \mid \overline{\ell} \mid \ell_1 \sqcup \ell_2 \mid \ell_1 \sqcap \ell_2 \\
\textit{(integer expressions)} & exp & ::= & x \mid n \mid exp_1 \,\mathbf{op}\, exp_2 \mid \mathbf{read}\, x \\
\textit{(expressions)} & e & ::= & exp \mid \ell \\
\textit{(commands)} & cmd & ::= & \mathbf{skip} \mid (x_1, \ldots, x_n) := (e_1, \ldots, e_n) \mid \\
& & & \mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end} \mid cmd_1; cmd_2 \mid \\
& & & \mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end} \mid \mathbf{send}\ x_1\ \mathbf{to}\ x_2 \mid \\
& & & \mathbf{if}\, \ell_1 \sqsubseteq \ell_2\, \mathbf{then}\, (\mathbf{send}\, x_1\, \mathbf{to}\, x_2)\, \mathbf{else\ fail\ end}
\end{array}
$$

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. In the target language, only security levels $L$ and $H$ are represented at runtime. The security level $U$ used in the source language typing is replaced by variables and expressions on variables. Level expressions support operators for supremum, infimum and complement (where $\overline{L} = H$ and $\overline{H} = L$); these are defined in Sect. 4.3.

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 $m_v : (\mathcal{V}_{level} \rightarrow \{L, H\}) \uplus (\mathcal{V} \setminus \mathcal{V}_{level} \rightarrow \mathbb{Z} \uplus \mathcal{C})$. Furthermore we assume that $\mathcal{V}_{level}$ contains variables $\_\mathsf{pc}$ and $\_\mathsf{hc}$, and, for each variable $x \in \mathcal{V} \setminus \mathcal{V}_{level}$ there exist level variables $x_{\mathsf{lev}}$; for channel variables, we also have a level variable for their content, that is, the level of the information stored in the channel that the variables point to, written $x_{\mathsf{ch}}$. They will be used in instrumented programs to track security levels. For example, if $x$ is a channel variable of security type $(int_\ell\ chan)_{\ell'}$, then the values of these variables should be $x_{\mathsf{ch}} = \ell$ and $x_{\mathsf{lev}} = \ell'$ (this will be ensured by our instrumentation). Variables _pc and _hc hold the security levels of the context and halting context respectively. What these are will be explained in Sect. 4.3. Note that the simultaneous assignment $(x_1, \ldots, x_n) := (e_1, \ldots, e_n)$ is introduced to ensure coherence between the value of a label variable and the level of the value assigned to the corresponding variable. For all other common commands, the semantics of the target language is the same as in the source language.

## 4.3 Instrumentation as a Type System

Our instrumentation algorithm is specified as a type system in Fig. 2. Its primary goal is to inline monitor actions in the program under analysis, thereby generating a safe version of it. Its secondary goal is to reject programs that contain obvious leaks of information. The inlined actions are essentially updates and checks of level variables to prevent a **send** command from leaking information.

The typing rules of variables and constants have judgements of the form $\Gamma \vdash e : \sigma_\ell$, telling that $\sigma_\ell$ is the variable type of $e$. The instrumentation judgements are of the form $\Gamma, pc, hc \vdash cmd : t, h, \Gamma', [\![cmd]\!]$ where $\Gamma, \Gamma' : \mathcal{V} \uplus \mathcal{C} \rightarrow VarT$ 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 cause 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 five *termination types* $\mathcal{T} = \{T, D, M_L, M_U, M_H\}$, where $T$ means that a command terminates for all memories, $D$, diverges for all memories, $M_L$, $M_H$ and $M_U$ mean that a command's termination is unknown statically; the subscript is used to indicate on which level the termination depends. For example, the termination of the loop in Listing 1.2 is $M_H$ because it can either terminate or diverge at runtime, and this depends on information of level $H$. The loop in Listing 1.3 on the other hand is of termination type $T$ because, no matter what the value of `highValue` is, it will always eventually terminate. Similarly, a loop whose condition is always true will have termination type $D$ since it always diverges. The precision of this analysis depends on the oracle precision.

The instrumentation of a program $p$ begins by inserting commands to initialize a few level variables: _pc, _hc are initialized to $L$, as well as the level variables $x_{\mathsf{lev}}$ and $x_{\mathsf{ch}}$ for each variable $x \in \mathcal{V}$ appearing in $p$. Similarly, level variables $c_{\mathsf{lev}}$ and $c_{\mathsf{ch}}$ associated with each channel $c$ used in $p$ are also initialized, but the latter rather gets initialized to $levelOfChan(c)$. After initialization, instrumentation is given by the rules of Fig. 2. We now explain these rules.

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

$$\text{(S-Op)}\ \frac{\Gamma \vdash e_1 : \mathit{int}_{\ell_1} \qquad \Gamma \vdash e_2 : \mathit{int}_{\ell_2}}{\Gamma \vdash e_1\ \mathbf{op}\ e_2 : \mathit{int}_{\ell_1 \sqcup \ell_2}} \qquad \text{(S-Skip)}\ \Gamma, pc, hc \vdash\ \mathbf{skip} :\ T, hc, \Gamma, \mathsf{skip}$$

$$\text{(S-Assign)}\ \frac{\Gamma \vdash e : \sigma_{\ell_e}}{\Gamma, pc, hc \vdash x := e :\ T, hc, \Gamma[x \mapsto \sigma_{pc \sqcup \ell_e}], \mathsf{genassign}}$$

$$\text{(S-Send)}\ \frac{\Gamma(x) = \mathit{int}_{\ell_x} \qquad \Gamma(c) = (\mathit{int}_\ell\ \mathit{chan})_{\ell_c} \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell_c) \sqsubseteq_m \ell}{\Gamma, pc, hc \vdash \mathbf{send}\ x\ \mathbf{to}\ c : T, hc \sqcup \ell_c, \Gamma, \mathsf{gensend}}$$

$$\text{(S-If)}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathit{int}_{\ell_e} \qquad h_3 = \sqcup_{j \in \{1,2\}} d(\Gamma, pc \sqcup \ell_e, \mathit{cmd}_j) \\ \bot \notin \mathrm{ran}(\Gamma_1 \sqcup \Gamma_2) \qquad h = (h_1 \sqcup h_2 \sqcup h_3 \sqcup \mathit{level}(t_1 \oplus_{\ell_e} t_2)) \\ \Gamma, pc \sqcup \ell_e, hc \vdash \mathit{cmd}_j : t_j, h_j, \Gamma_j, [\![\mathit{cmd}_j]\!] \quad j \in \{1,2\}\end{array}}{\Gamma, pc, hc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ \mathit{cmd}_1\ \mathbf{else}\ \mathit{cmd}_2\ \mathbf{end} : (t_1 \oplus_{\ell_e} t_2), h, \Gamma_1 \sqcup \Gamma_2, \mathsf{genif}}$$

$$\text{(S-Loop)}\ \frac{\begin{array}{c}O(e, \mathit{cmd}, \Gamma \sqcup \Gamma') = t_o \qquad h = d(\Gamma, pc \sqcup \ell_e, \mathit{cmd}) \\ \ell_t = \mathit{level}(t) \qquad \ell_o = \mathit{level}(t_o) \qquad \bot \notin \mathrm{ran}(\Gamma \sqcup \Gamma') \qquad \Gamma \sqcup \Gamma' \vdash e : \mathit{int}_{\ell_e} \\ \Gamma \sqcup \Gamma', (pc \sqcup \ell_e), (hc \sqcup \ell_t \sqcup h') \vdash \mathit{cmd} : t, h', \Gamma', [\![\mathit{cmd}]\!]\end{array}}{\Gamma, pc, hc \vdash \mathbf{while}\ e\ \mathbf{do}\ \mathit{cmd}\ \mathbf{end} : t_o, h \sqcup h' \sqcup \ell_o, \Gamma \sqcup \Gamma', \mathsf{genwhile}}$$

$$\text{(S-Seq1)}\ \frac{\Gamma, pc, hc \vdash \mathit{cmd}_1 : D, h, \Gamma_1, [\![\mathit{cmd}_1]\!]}{\Gamma, pc, hc \vdash \mathit{cmd}_1; \mathit{cmd}_2 : D, h, \Gamma_1, [\![\mathit{cmd}_1]\!]}$$

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

**Fig. 2.** Instrumentation and typing rules for the source language

Rules (S-Chan) and (S-Int) specify the channels type and integer constants. Rule (S-Var) encodes the typing of a variable, as given by environment $\Gamma$. Rule (S-Op) encodes expression typing and excludes channel operations. Rule (S-Read) specifies the current $c$ value type. To prevent implicit flows, the specified security level takes into account the assignment context of channel variable $c$, hence the supremum $\ell \sqcup \ell_c$ Rule (S-Assign) specifies the type of $x$ from the one of $e$ to prevent explicit flows, and from $pc$, to prevent implicit flows. Its instrumentation is given by the following macro:

$$\mathsf{genassign} = \begin{cases}(x, x_{\mathsf{lev}}) := (e, \_\mathsf{pc} \sqcup e_{\mathsf{lev}}) & \text{if } \sigma = \mathit{int} \\ (x, x_{\mathsf{lev}}, x_{\mathsf{ch}}) := (e, \_\mathsf{pc} \sqcup e_{\mathsf{lev}}, e_{\mathsf{ch}}) & \text{if } \sigma = \mathit{int}_{\ell'}\, \mathit{chan}\end{cases}$$

The variable $e_{\mathsf{lev}}$ represents the level of expression $e$, as specified by Rule (S-Op). For example if $e = x + \textbf{read}\ c$, then $e_{\mathsf{lev}} = x_{\mathsf{lev}} \sqcup c_{\mathsf{ch}} \sqcup c_{\mathsf{lev}}$. If $e = x + y$ then $e_{\mathsf{lev}} = x_{\mathsf{lev}} \sqcup y_{\mathsf{lev}}$. Rule (S-Send) requires $(pc \sqcup hc \sqcup \ell_x \sqcup \ell_c) \sqsubseteq_m \ell$. The four variables on the left-hand side correspond to the information level possibly revealed by the output to $x_2$. The instrumentation translates it as follows

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

The halting context records the possible failure of the guarded send, it is updated with the assignment context of the channel. The following example illustrates why this is necessary.

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

Listing 1.4: Dangerous runtime halting

Assume that `unknownValue` is private 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`. Updating $\_\mathsf{hc}$ will affect the check of all subsequent guarded send. Updating $\_\mathsf{hc}$ with $x_{\mathsf{lev}}$ or $\_\mathsf{pc}$ is not necessary since their value will be the same for all low-equivalent memories.

For the conditional rules, we need a union of environments that maps to each variable appearing in both branches the supremum of the two variable types, and for each channel variable appearing in both branches the security level $U$ if the levels of their content differ.

**Definition 3.** *The supremum of two environments is given as $dom(\Gamma_1 \sqcup \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$, and*

$$(\Gamma_1 \sqcup \Gamma_2)(x) = \begin{cases} \Gamma_i(x) & \text{if } x \in dom(\Gamma_i) \backslash dom(\Gamma_j), \{i,j\} = \{1,2\} \vee \Gamma_1(x) = \Gamma_2(x) \\ (int_U\, chan)_{\ell_2 \sqcup \ell_2'} & \text{if } \Gamma_1(x) = (int_{\ell_1}\ chan)_{\ell_2} \\ & \wedge \Gamma_2(x) = (int_{\ell_1'}\ chan)_{\ell_2'} \wedge \ell_1 \neq \ell_1' \\ \sigma_{\ell \sqcup \ell'} & \text{if } \Gamma_1(x) = \sigma_\ell \wedge \Gamma_2(x) = \sigma_{\ell'} \\ \bot & \text{otherwise.} \end{cases}$$

The symbol $\bot$ is used to indicate that a typing inconsistency occured, e.g. when a variable is used as an integer in one branch and as a channel in another.

The function $level : \mathcal{T} \to \mathcal{L}$ returns the *termination level* (i.e., the level that 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.

$$t_1 \oplus_\ell t_2 = \begin{cases} t_1 & \text{if } t_1 = t_2 \wedge [t_1 \neq M_L \vee \ell = L] \\ M_L & \text{if } \ell = L \wedge t_1 \neq t_2 \wedge \{t_1, t_2\} \subseteq \{T, D, M_L\} \\ M_H & \text{if } \ell = H \wedge [M_{\ell'} \in \{t_1, t_2\}, \ell' \in \mathcal{L} \text{ or } \{t_1, t_2\} = \{T, D\}] \\ M_{\langle \ell \sqcup \ell_1 \sqcup \ell_2 \rangle} & \text{otherwise, } t_1 = M_{\langle \ell_1 \rangle}, t_2 = M_{\langle \ell_2 \rangle} \end{cases}$$

where $\langle e \rangle$ is $e$, a level expression, without evaluation. We will evaluate $\langle e \rangle$ to $U$ in the instrumentation type system (Fig. 2). We prefer to write $\langle e \rangle$ to emphasise the fact that $U$ is the approximation of an expression.

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

The following example shows one more requirement. If in Listing 1.5 variable `unknownChannel` is a public channel at runtime, and 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 1.5: A guarded send can generate a progress leak

The following function $d : ((\mathcal{V} \uplus \mathcal{C} \rightarrow \mathit{VarT}) \times \mathcal{L} \times \mathit{Cmd}) \rightarrow \mathcal{L}$, is used to update the halting context, where $\mathit{Cmd}$ is the set of commands. It approximates the information level that could be leaked through progress channels by a possibly failed guarded send in an unexecuted branch. Here, if $\Gamma(c) = (\mathit{int}_\ell\ \mathit{chan})_{\ell'}$, then we write $\Gamma_{\mathsf{ch}}(c) = \ell$ and $\Gamma_{\mathsf{lev}}(c) = \ell'$.

$$d(\Gamma, pc, cmd) = \begin{cases} pc \sqcap \big( \bigsqcup_{c \in dc} (\overline{\Gamma_{\mathsf{ch}}(c)} \sqcup \Gamma_{\mathsf{lev}}(c)) \big) & \text{if } dc \cap mv = \emptyset \\ pc & \text{otherwise} \end{cases}$$

In this definition, $dc$ represents the set of *dangerous channels*, that is, the ones appearing in at least one guarded send in $[\![cmd]\!]$; $mv$ is the set of variables that may be modified in $cmd$. Intuitively, if all the dangerous channels are of level $H$ and not modified inside $cmd$, then we know that these guarded send cannot fail. If we cannot be sure of their level, then the halting context is updated with level $pc$. The supremum over the security levels of channels is taken in case the value of the channels is sensible (for example if `lowValue` was H in Listing 1.1).

(S-IF) Its instrumentation is given by the following macro:

```
genif = _oldpc^ν := _pc;          where insIF(i,other) = _pc := _pc ⊔ e_lev;
        if e                                           hd(_hif^ν, mv_other, dc_other);
          then insIF(1,2)                              [[cmd_i]];
          else insIF(2,1)                              _hc := _hc ⊔ _hif;
        end;                                           uphc(t_1, t_2, e_lev);
        _pc := _oldpc                                  update(mv_other)
```

where $dc_j$ represents the set of channels appearing in at least one **guardedSend** in $[\![cmd_j]\!]$, $mv_j$ is the set of variables that may be modified in $cmd_j$, $t_j$ is the termination type of $cmd_j$, $e_{\mathsf{lev}}$ is the guard condition's level expression and $\ell_e$ is the level of this guard (as computed by the typing system).

The instrumented code starts by saving the current context to $\_\mathsf{oldpc}^\nu$ (the symbol $\nu$ 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. In each branch, function hd, function $d$'s at runtime, evaluates the information level possibly revealed by a failed guarded send in the other branch.

$$\mathsf{hd}(\_\mathsf{h}, mv, dc) = \begin{cases} \_\mathsf{h} := (\_\mathsf{pc} \sqcap (\bigsqcup_{c \in dc} \overline{c_{\mathsf{ch}}} \sqcup c_{\mathsf{lev}})) & \text{if } dc \cap mv = \emptyset \\ \_\mathsf{h} := \_\mathsf{pc} & \text{otherwise} \end{cases}$$

This must be computed before executing $[\![cmd_j]\!]$ because we want to evaluate whether the untaken branch could halt the execution or not. This must be done before $[\![cmd_j]\!]$ as the latter could modify the level of the dangerous channels.

Function uphc is used to generate the code updating the halting context.

$$\mathsf{uphc}(t_1, t_2, e_{\mathsf{lev}}) = \begin{cases} \mathsf{skip} & \text{if } t_1 = t_2 \in \{T, D\} \\ \_\mathsf{hc} := \_\mathsf{hc} \sqcup e_{\mathsf{lev}} & \text{otherwise} \end{cases}$$

The rational underlying uphc use is to protect the guard value from being revealed. If we know that both branches behave similarly, then the adversary will not be able to deduce private information. On the other hand, if the two branches may not behave the same way, then we have to perform $\_\mathsf{hc} := \_\mathsf{hc} \sqcup e_{\mathsf{lev}}$.

The following function updates the level variables of the untaken branch's modified variables so that they have similar types in all low-equivalent memories.

$$\mathsf{update}(mv) = \begin{cases} \mathsf{skip} & \text{if } mv = \emptyset \\ (x, x_{\mathsf{lev}}) := (x, x_{\mathsf{lev}} \sqcup \_\mathsf{pc}); \mathsf{update}(mv \backslash \{x\}) & \text{if } x \in mv. \end{cases}$$

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

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

Listing 1.6: Example illustrating why it is necessary to update the modified variables

(S-Loop) Typing the **while** involves a fixed point computation due to the flow sensitivity. It is easy to show that this computation converges. The typing relies on $O$, a statically called oracle computing the termination loop type ($t_o$).

```
genwhile = _oldpc^ν := _pc;              where iWhile =   _pc := _pc ⊔ e_lev;
           iWhile;                                         update(mv);
           while e do                                      hd(_hwhile^ν, mv, dc);
             ⟦cmd⟧                                         _hc := _hc ⊔ _hwhile;
             iWhile; end;                                  uphc(t_o, t_o, e_lev);
           _pc := _oldpc
```

The inserted commands are similar to those of the **if**. The level variables and halting context are updated before the loop in case an execution does not enter the loop. They must be updated at the end of each iteration for the next iteration.

(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 $h_1$.

In a longer version, we present a type system for the target language. We show that a well typed program satisfies the progress-sensitive non-interference property 1, and that a program generated by our typing system is well typed.

## 5 Related Work

There has been much research in language-based techniques for controlling information flow over the last two decades.

Le Guernic et al. [8] 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 but it takes into account concurrency. In Russo and Sabelfeld [11], the authors 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.

Moore et al. [9] 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 [5] inline a flow-sensitive progress-insensitive hybrid monitor and prove soundness by bisimulation. We inline a flow-sensitive progress-sensitive hybrid monitor, and we prove soundness using a mostly-standard security-type system for the target language.

Askarov and Sabelfeld [3] use hybrid monitors to enforce information security in dynamic languages based on on-the-fly static analysis. They provide a model to define non-interference that is suitable to progress-sensitivity and they quantify information leaks due to termination [2].

Askarov et al. [1] introduce a progress-sensitive hybrid monitoring framework where the focus is on concurrent programs, and the use of rely-guarantee reasoning to enable fine-grained sharing of variables between threads. Each thread is guarded by its own local monitor (progress- and flow-sensitive). Their local monitor could be replaced by a variant of our inlined monitor.

## 6 Conclusion

We have presented a hybrid information flow enforcement mechanism in which the main contributions are the following.

(a) Our monitor is one of the first hybrid monitor that is both flow- and progress-sensitive. It is more precise and introduces less overhead than currently available solutions (e.g., [9,10]). Since our monitor is inlined, it can be easily optimized using classical partial evaluation techniques, [7].

(b) We solve a few issues such as (1) the fact that it is not possible to approximate the level of a channel (by introducing a level $U$) and (2) the need to approximate the level of information that could be leaked through progress channels (by introducing a function $d$).

We believe our approach to be generalizable to complex lattices, but it will require a few alterations. Instead of only one uncertain level $U$, we would use sets of possible levels ($U$ is, in some sense, an abstraction of the set $\{L, H\}$) that are ordered pointwise. That is, $\{L\} \sqsubseteq \{L, H\} \sqsubseteq \{H\}$. The function $d$ would have to be adapted. Namely, the complement operation in $d$ would have to be replaced with the following expression: $\{\ell : \ell \not\sqsubseteq \Gamma_{\mathsf{ch}}(c)\} \cap \{\ell : \ell \not\sqsupset pc\}$.

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.

## References

1. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: Computer Security Foundations Symposium (2015)
2. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proceedings of the European Symposium on Research in Computer Security: Computer Security (2008)
3. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: CSF (2009)
4. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the Workshop on Programming Languages and Analysis for Security (2009)
5. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: Proceedings of the 23rd IEEE Security Foundations Symposium (2010)
6. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)
7. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall, Englewood Cliff (1993)
8. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-based confidentiality monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)
9. Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: CCS 2012 (2012)
10. O'Neill, K.R., Clarkson, M.R., Chong, S.: Information-flow security for interactive programs. In: CSFW. IEEE (2006)
11. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: CSF, pp. 186–199. IEEE Computer Society (2010)

12. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
13. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: POPL (1998)
14. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2), 167–187 (1996)
15. Zheng, L., Myers, A.C.: Dynamic security labels and noninterference. In: Dimitrakos, T., Martinelli, F. (eds.) Formal Aspects in Security and Trust. IFIP, vol. 173. Springer, Heidelberg (2005)