Enforcing information flow policies by a three-valued analysis

Josée Desharnais, Erwanne P. Kanyabwero, and Nadia Tawbi

Department of Computer Science and Software Engineering, Université Laval,
josee.deharnais@ift.ulaval.ca, erwamme-pamela.kanyabwero.1@ulaval.ca,
nadia.tawbi@ift.ulaval.ca

Abstract. This paper presents an approach to enforce information flow policies using a three-valued type-based analysis on a core imperative language. Our analysis aims first at reducing false positives generated by static analysis, and second at preparing for instrumentation. False positives arise in the analysis of real computing systems when some information is missing at compile time, for example the name of a file, and consequently, its security level. The key idea of our approach is to distinguish between negative and may responses. Instead of rejecting in the latter cases, we type instructions with an additional type, *unknown*, indicating uncertainty, possibly preparing for a light instrumentation. During the static analysis step, the may responses are identified and annotated with the *unknown* security type, while the positive and negative responses are treated as is usually done. This work is done in preparation of a hybrid security enforcement mechanism: the *maybe-secure* points of the program detected by our type-based analysis will be instrumented with dynamic tests. We prove that our type system is sound by showing that it satisfies non-interference. The novelty is the handling of three security types, but we also treat variables and channels in a special way. Programs interact via communication channels. Secrecy levels are associated to channels rather than to variables whose security levels change according to the information they store.

1 Introduction

Secure information flow analysis is a technique used to prevent misuse of data. This is done by restricting how data are transmitted among variables or other entities in a program, according to their security classes.

Our objective is to take advantage of the combination of static and dynamic analysis. We design a three-valued type system to statically check non-interference for a simple imperative programming language. To the usual two main security levels, public (or *Low*) and private (or *High*), we add a third value, *Unknown*, that captures the possibility that we may not know, before execution, whether the information is public or private. Standard two-valued analysis has no choice but to be pessimistic with uncertainty and hence generate false positive alarms. If uncertainty arises during the analysis, we tag the instruction in cause: in a second step, instrumentation at every such point together with dynamic analysis will allow us to head to a more precise result than purely static approaches. We get reduced false alarms, while introducing a light runtime overhead by instrumenting only when there is a need for it.

The goal of a security analysis is to ensure non-interference, that is, to prevent inadvertent information leaks from private channels to public channels. More precisely, in our case, the goal is to ensure that 1) a well-typed program satisfies non-interference, 2) a program not satisfying non-interference is rejected 3) a program that may satisfy non-interference is detected and sent to the instrumentation step. Furthermore, we consider that programs have interaction with an external environment through communication *channels*, i.e., objects through which a program can get information from users (printing screen, file, network, etc.). In contrast with the work of Volpano et al. [18], variables are not necessarily channels, they are local and hence their security type is allowed to change throughout the program. This is similar to flow-sensitive typing approaches like the one of Hunt and Sands, or Russo and Sabelfeld [7, 12]. Our approach distinguishes clearly communication channels, through which the program interacts and which have a priori security levels, from variables, used locally. Therefore, our definition of non-interference applies to communication channels: someone observing the final information contained in communication channels cannot deduce anything about the initial content of the channels of higher security level.

We aim at protecting against two types of flows, as explained in [5]: *explicit flow* occurs when the content of a variable is directly transferred to another variable, whereas *implicit flow* happens when the content assigned to a variable depends on another variable, i.e., the guard of a conditional structure.

The rest of this paper is organized as follows. After describing in Section 2 the programming language used, we present the type system ensuring that information will not be leaked improperly, in Section 3. The soundness of the type system is proved in Section 4. We compare our work to similar approaches in the literature in Section 5. We conclude in Section 6.

2 Programming language

We illustrate our approach on a simple imperative programming language, a variant of the one presented by Smith [16], which we adapt to deal with 3-valued security levels.

2.1 Syntax

Let $\mathcal{V}ar$ be a set of identifiers for variables, and $\mathcal{C}$ a set of communication channel names. Throughout the paper, we use generically the following notation: variables are $x \in \mathcal{V}ar$, and there are two types of constants: $n \in \mathbb{N}$ and $nch \in \mathcal{C}$. The syntax is as follows:

$$
\begin{array}{lll}
\textit{(instructions)} & p ::= & e \mid c \\
\textit{(expressions)} & e ::= & x \mid n \mid nch \mid e_1 \ \mathbf{op} \ e_2 \\
\textit{(commands)} & c ::= & \mathbf{skip} \mid x := e \mid c_1 ; c_2 \\
 & & \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \ \mathbf{end} \mid \mathbf{while} \ e \ \mathbf{do} \ c \ \mathbf{end} \mid \\
 & & \mathbf{receive_c} \ x_1 \ \mathbf{from} \ x_2 \mid \\
 & & \mathbf{receive_n} \ x_1 \ \mathbf{from} \ x_2 \mid \\
 & & \mathbf{send} \ x_1 \ \mathbf{to} \ x_2
\end{array}
$$

Instructions are either expressions or commands. Values are integers (we use zero for false and nonzero for true), or channel names. **op** stands for arithmetic or logic binary operators on integers and comparison operators on channel names. Commands are mostly the standard instructions of imperative programs.

We suppose that two programs can only communicate through channels (which can be, for example, files, network channels, keyboards, computer screens, etc.). We assume that the program has access to a pointer indicating the next element to be read in a channel and that the send to a channel would append an information in order for it to be read in a first-in-first-out order. When an information is read in a channel it does not disappear, only the read pointer is updated, the observable content of a channel remains as it was before. Our programming language is sequential; we do not claim to treat concurrency and communicating processes as it is treated in [10, 8]. We consider that external processes can only read and write to public channels. The instructions related to accessing channels deserve further explanations.

- $\mathbf{receive_c}$ x_1 **from** x_2: stands for "receive content". It represents an instruction that reads a value from a channel with name x_2 and assigns its content to x_1.
- $\mathbf{receive_n}$ x_1 **from** x_2: stands for "receive name". Instead of getting data from the channel, we receive another channel name, which might be used further in the program. This variable has to be treated like a channel.
- **send** x_1 **to** x_2: used to output on a channel with name x_2 the content of the variable x_1.

The need for two different receive commands is a direct consequence of our choice to distinguish variables from channels. It will be clearer when we explain the typing of commands, but observe that this allows, for example, to receive a private name of channel through a public channel [1]: the information can have a security level different from its origin's. This is not possible when variables are observable.

2.2 Semantics

The behavior of the program follows the structural operational semantics shown in Table 1. An instruction p is executed under a memory map $\mu : \mathcal{V}ar \rightarrow \mathbb{N} \cup \mathcal{C}$. Hence the semantics specifies how *configurations* $\langle p, \mu \rangle$ evolve, either to a value,

[1] but not the converse, to avoid implicit flow leaks

(VAL)	$\langle v, \mu \rangle \rightarrow_e v,$
(VAR)	$\langle x, \mu \rangle \rightarrow_e \mu(x),$
(OP)	$\dfrac{\langle e_1, \mu \rangle \rightarrow_e v_1 \qquad \langle e_2, \mu \rangle \rightarrow_e v_2 \qquad v_1 \ \mathbf{op} \ v_2 = n}{\langle e_1 \ \mathbf{op} \ e_2, \mu \rangle \rightarrow_e n}$
(SKIP)	$\langle \mathbf{skip}, \mu \rangle \rightarrow \mu$
(ASSIGN)	$\dfrac{\langle e, \mu \rangle \rightarrow_e v}{\langle x := e, \mu \rangle \rightarrow \mu[x \mapsto v]}$
(RECEIVE-CONTENT)	$\dfrac{x_2 \in \mathit{dom}(\mu) \qquad \mathit{read}(\mu(x_2)) = n}{\langle \mathbf{receive_c} \ x_1 \ \mathbf{from} \ x_2, \mu \rangle \rightarrow \mu[x_1 \mapsto n]}$
(RECEIVE-NAME)	$\dfrac{x_2 \in \mathit{dom}(\mu) \qquad \mathit{read}(\mu(x_2)) = \mathit{nch}}{\langle \mathbf{receive_n} \ x_1 \ \mathbf{from} \ x_2, \mu \rangle \rightarrow \mu[x_1 \mapsto \mathit{nch}]}$
(SEND)	$\dfrac{x_1 \in \mathit{dom}(\mu)}{\langle \mathbf{send} \ x_1 \ \mathbf{to} \ x_2, \mu \rangle \rightarrow \mu, \mathit{update}(\mu(x_2), \mu(x_1))}$
(CONDITIONAL)	$\dfrac{\langle e, \mu \rangle \rightarrow_e n \qquad n \neq 0}{\langle \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \ \mathbf{end}, \mu \rangle \rightarrow \langle c_1, \mu \rangle}$
	$\dfrac{\langle e, \mu \rangle \rightarrow_e n \qquad n = 0}{\langle \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \ \mathbf{end}, \mu \rangle \rightarrow \langle c_2, \mu \rangle}$
(LOOP)	$\dfrac{\langle e, \mu \rangle \rightarrow_e n \qquad n = 0}{\langle \mathbf{while} \ e \ \mathbf{do} \ c \ \mathbf{end}, \mu \rangle \rightarrow \mu}$
	$\dfrac{\langle e, \mu \rangle \rightarrow_e n \qquad n \neq 0}{\langle \mathbf{while} \ e \ \mathbf{do} \ c \ \mathbf{end}, \mu \rangle \rightarrow \langle c; \mathbf{while} \ e \ \mathbf{then} \ c \ \mathbf{end}, \mu \rangle}$
(SEQUENCE)	$\dfrac{\langle c_1, \mu \rangle \rightarrow \mu'}{\langle c_1; c_2, \mu \rangle \rightarrow \langle c_2, \mu' \rangle}$

Table 1. Structural operational semantics

another configuration, or a memory. Evaluation of expressions under a memory involves no "side effects" that would change the state of memory. In contrast, the role of commands is to be executed and change the state. Thus we have two evaluation rules: $\langle e, \mu \rangle$ leads to a value resulting from the evaluation of expression e on memory μ; this transition is designated by $\rightarrow_e$, $\langle c, \mu \rangle$ leads to a memory produced by the execution of command c on memory μ; this transition is designated by $\rightarrow$.

skip leaves the memory state unchanged. The assignment $x := e$ results in a memory identical to μ, except that its value at variable x is now the evaluation of e.

$\mathbf{receive_c}$ x_1 **from** x_2 and $\mathbf{receive_n}$ x_1 **from** x_2 are semantically evaluated similarly. Information from the channel x_2 is read and assigned to the variable x_1. The distinctive feature of the rule RECEIVE-CONTENT is that the result of evaluation is an integer variable, while for the rule RECEIVE-NAME, the result is a channel name. Here, we introduce a generic function *read(channel)* that represents the action of getting information from a channel (eg. get a line from a file, input from the keyboard, etc.). The content of a channel remains the same after both kind of receive.

send x_1 **to** x_2 updates the channel x_2 with the value of the variable x_1. This is done by the generic function *update(channel, information)*, which represents the action of updating the channel with some information. Note that the content of the variable x_2, that is, the name of the channel, does not change; hence μ stays the same. The content of the channel is updated after a **send**.

A conditional statement **if** e **then** c_1 **else** c_2 **end** evaluates to c_1 or c_2 depending whether e evaluates to **true** (nonzero) or **false** (zero).

For the loop rule, if the boolean condition evaluates to **false**, the loop is not entered and the memory remains the same; if the condition evaluates to *true*, the body c is executed followed sequentially by the re-execution of the **while** instruction.

If c_1 transforms memory μ into μ', then $c_1; c_2$ amounts to the execution of c_2 on μ'.

3 Security type system

We now present the security type system that we use to check whether a program, written in the language described above, either satisfies non-interference, may satisfy it or does not satisfy it. The security types are defined as follows:

$$\begin{array}{ll} \textit{(data types)} & \tau ::= \ L \mid U \mid H \\ \textit{(instruction types)} & \rho ::= \tau \ \mathit{val} \mid \tau \ \mathit{chan} \mid \tau \ \mathit{cmd} \end{array}$$

We consider a set of three security levels $SL = \{L, U, H\}$. This set is extended to a lattice $(SL, \sqsubseteq)$ using the following order: $L \sqsubseteq U \sqsubseteq H$ (we use freely the usual symbols $\sqsupseteq$ and $\sqsupset$). It is with respect to this order that the supremum $\sqcup$ and infimum $\sqcap$ over security types are defined. We lift this order to instruction types and maps in the trivial way and assume these operation return $\perp$ when applied to instructions of different types, e.g., $H\ chan \sqcup H\ val = \perp$. We also need the following weaker relation $\preceq$, defined as:

$$x \npreceq y \ \text{ iff } \ x = H \text{ and } y = L.$$

This relation can be interpreted as "maybe $\sqsubseteq$": it is used to ensure that rejection of a program will only occur if there is a flow from H to L; it will be explained later on.

A label is added to the type of an instruction in order to indicate whether the information is an integer value, a channel name or a command. When typing a program, security types are assigned to variables, channels and commands – and to the context of execution. The meaning of types is as follows. A variable of type τ *val* has a content of security type τ; a channel of type τ *chan* can store information of type τ or lower (indeed, a private channel must have the possibility to contain or receive both private and public information). The security typing of commands is standard, but has a slightly different meaning: a command of type τ *cmd* is guaranteed to only allow flows into channels whose security types are τ or higher. Hence, if a command is of type L *cmd* then it may contain a flow to a channel of type L *chan*.

Our type system has two interesting properties: *simple security* applying to expressions and *confinement* applying to commands [16]. *Simple security* says that an expression e of type τ *val* or τ *chan* contains only variables of level τ or lower. Simple security ensures that the type of a variable is consistent with the principle stated in the precedent paragraph. *Confinement* says that a command c of type τ *cmd* executed under a context of type *pc* allows flows only to channels of level $\tau \sqcup pc$ or higher, in order to avoid a flow from a channel to another of lower security (H to L for example). Those two properties are used to prove non-interference.

Our typing rules are shown in Table 2. A *typing judgment* has the form $\Gamma, pc \vdash p : \rho, \Gamma'$, where Γ and Γ' are typing environments, mapping variables to a type of the form τ *val* or τ *chan*, representing their security level; *pc* is the security type of the context. The program is typed with a context of type L; according to the security types of conditions, some blocks of instructions are typed with a higer context, as will be explained later. The typing judgment can be read as: within an initial typing environment Γ and a security type context *pc*, the command p has type ρ, yielding a final environment Γ'. When the typing environment stays unchanged, Γ' is omitted. Since the type of channels is constant, there is a particular typing environment for channel constants, named *TypeOf_Channel* that is given before the analysis. In the rules, α stands for either the label *val* or *chan*, depending on the context.

There are three operators on typing environments that we need to define: $\Gamma \dagger [x \mapsto \rho]$, $\Gamma \sqcup \Gamma'$ and $\overline{\Gamma}$. The former is a standard update, where the image of x is set to ρ, no matter if x is in the original domain of Γ or not. For the conditional rule, we need to perform a union of environments where common value variables must be given, as security type, the supremum of the two types, and where channel variables are given type U if they differ. More precisely, we

(CHAN_S)	$\dfrac{TypeOf\_Channel(nch) = \tau}{\Gamma, pc \vdash nch : \tau\ chan}$	(INT_S)	$\Gamma, pc \vdash n : L\ val$
(OP_S)	$\dfrac{\Gamma, pc \vdash e_1 : \tau_1\,\alpha, \qquad \Gamma, pc \vdash e_2 : \tau_2\,\alpha}{\Gamma, pc \vdash e_1\ \mathbf{op}\ e_2 : (\tau_1 \sqcup \tau_2)\ val}$	(VAR_S)	$\dfrac{\Gamma(x) = \tau\,\alpha}{\Gamma, pc \vdash x : \tau\,\alpha}$
(SKIP_S)	$\Gamma, pc \vdash\ \mathbf{skip}\ :\ H\ cmd$		
(ASSIGN -VAL_S)	$\dfrac{\Gamma, pc \vdash e : \tau\ val}{\Gamma, pc \vdash x := e : (\tau \sqcup pc)\ cmd, \Gamma \dagger [x \mapsto (\tau \sqcup pc)\ val]}$		
(ASSIGN -CHAN_S)	$\dfrac{\Gamma, pc \vdash e : \tau\ chan \qquad pc \preceq \tau}{\Gamma, pc \vdash x := e : \tau\ cmd, \Gamma \sqcup [\_instr \mapsto \tau \sqcup pc] \dagger [x \mapsto \tau\ chan]}$		
(RECEIVE-VAL_S)	$\dfrac{\Gamma(x_2) = \tau\ chan}{\Gamma, pc \vdash \mathbf{rec_c}\ x_1\ \mathbf{from}\ x_2 : (\tau \sqcup pc)\ cmd, \Gamma \dagger [x_1 \mapsto (\tau \sqcup pc)\ val]}$		
(RECEIVE-NAME_S)	$\dfrac{\Gamma(x_2) = \tau\ chan \qquad pc \preceq \tau}{\Gamma, pc \vdash \mathbf{rec_n}\ x_1\ \mathbf{from}\ x_2 : \tau\ cmd, \Gamma \sqcup [\_instr \mapsto \tau \sqcup pc] \dagger [x_1 \mapsto U\ chan]}$		
(SEND_S)	$\dfrac{\begin{array}{c}\Gamma(x_1) = \tau_1\,\alpha, \\ \Gamma(x_2) = \tau\ chan, \qquad (\tau_1 \sqcup pc) \preceq \tau\end{array}}{\Gamma, pc \vdash \mathbf{send}\ x_1\ \mathbf{to}\ x_2 : \tau\ cmd, \overline{\Gamma}}$		
(COND_S)	$\dfrac{\begin{array}{c}\Gamma, (pc \sqcup \tau_0) \vdash c_1 : \tau_1\ cmd, \Gamma' \\ \Gamma, pc \vdash e : \tau_0\ val \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_2 : \tau_2\ cmd, \Gamma'' \qquad \Gamma' \sqcup \Gamma'' \sqsupset \bot\end{array}}{\Gamma, pc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end} : (\tau_1 \sqcap \tau_2)\ cmd, \Gamma' \sqcup \Gamma''}$		
(LOOP1_S)	$\dfrac{\Gamma, pc \vdash e : \tau_0\ val \qquad \Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ cmd, \Gamma' \qquad \Gamma = \Gamma \sqcup \Gamma' \sqsupset \bot}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau\ cmd\ \Gamma \sqcup \Gamma'}$		
(LOOP2_S)	$\dfrac{\begin{array}{c}\Gamma, pc \vdash e : \tau_0\ val \quad \Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ cmd, \Gamma' \quad \Gamma \neq \Gamma \sqcup \Gamma' \sqsupset \bot \\ \Gamma \sqcup \Gamma', (pc \sqcup \tau_0) \vdash \mathbf{while}\,e\,\mathbf{do}\,c\,\mathbf{end} : \tau'\ cmd, \Gamma''\end{array}}{\Gamma, pc \vdash \mathbf{while}\,e\ \mathbf{do}\,c\,\mathbf{end} : \tau'\ cmd,\ \Gamma''}$		
(SEQUENCE_S)	$\dfrac{\Gamma, pc \vdash c_1 : \tau_1\ cmd, \Gamma' \qquad \Gamma', pc \vdash c_2 : \tau_2\ cmd, \Gamma''}{\Gamma, pc \vdash c_1; c_2 : (\tau_1 \sqcap \tau_2)\ cmd, \Gamma''}$		

Table 2. Typing rules

extend $\sqcup$ to environments as follows: $\mathrm{dom}(\Gamma \sqcup \Gamma') = \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$, and

$$\Gamma \sqcup \Gamma'(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma') \\ \Gamma'(x) & \text{if } x \in \mathrm{dom}(\Gamma') \setminus \mathrm{dom}(\Gamma) \\ U & \text{if } \Gamma(x) = \tau\ chan \neq \tau'\ chan = \Gamma'(x) \\ \Gamma(x) \sqcup \Gamma'(x) & \text{otherwise.} \end{cases}$$

Note that $\Gamma \sqcup \Gamma'(x)$ can return $\perp$ if Γ and Γ' are incompatible on variable x, for example if $\Gamma(x)$ is a value, and $\Gamma'(x)$ is a channel (this can only happen if Γ and Γ' come from different branches of an **if** command).

The last operator on typing environment that we need to define relates to instrumentation. We introduce a special variable *_instr* whose type (maintained in the typing environment map) tells whether or not the program needs instrumentation. If the image of *_instr* is U or H then instrumentation is needed and L otherwise. Initially, $\Gamma(\_instr) = L$. Its value is updated to U or H through the supremum operator in rule RECEIVE-NAME_S and through an operator denoted by " $\overline{}$ ", in rule SEND_S, defined as follows:

$$\overline{\Gamma} = \Gamma \dagger [\_instr \mapsto U] \text{ whenever } (\tau_1 \sqcup pc) = \tau = U \text{ or } (\tau_1 \sqcup pc) \not\sqsubseteq \tau.$$

Before explaining each rule in details, let us give more explanation on $\preceq$, which occurs in rule SEND_S, for example. As said earlier, this relation is used to ensure that rejection of a program will only occur if there is a flow from H to L; During type analysis, we distinguish between safe flow programs and uncertain ones. For example, flows from U to H or L to U are secure because no matter what the types of uncertain variables actually are at runtime (L or H), the flows will always be secure. However, depending on the actual type of the U variable at runtime, a flow like U to L or U to U may be secure or not. A conservative analysis would reject a program with such flows but ours will tag the program as needing instrumentation and will carry on the type analysis. Consider the typing rule of the **send** instruction: the sending of x_1 on channel x_2 is accepted by the type system if $(\tau_1 \sqcup pc) \preceq \tau$, where τ_1 (resp. τ) is the type of x_1 (resp. x). This implies, by definition of $\preceq$, that if, for example, x_1 has an H content – or if the context is high – while x_2 is an L channel, then the rule cannot be applied, and consequently, the program will be rejected. Let us see how it works in other cases. Suppose that $(\tau_1 \sqcup pc) \preceq \tau$ and $(\tau_1 \sqcup pc) \sqsubseteq \tau$ but $\neg((\tau_1 \sqcup pc) = \tau = U)$; then the image of *_instr* under Γ is unchanged. This means that the flow from x_1 to x_2 is safe without any doubt. However, in all remaining cases, i.e., H to U, U to L and U to U, there is a risk of information leakage, and hence $\overline{\Gamma}(\_instr) = U$ indicating a need for instrumentation. The reason why we use a particular relation $\preceq$ instead of $\sqsubseteq$ is to allow the typing to progress until the end in case of uncertainty. If we had used the usual order relation $\sqsubseteq$, flows from H to U and U to L would have been rejected, even if there is a possibility that the variable of type U turns out to be a secure one. Handling this uncertainty is the main contribution of this paper and allows to reduce the number of false positive.

In related work, there are *subtyping judgements* of the form $\rho_1 \subseteq \rho_2$ or $\rho_1 \leq \rho_2$ [16, 18]. For instance, given two security types τ and τ', if $\tau \subseteq \tau'$ then any data of type τ can be treated as data of type τ'. Similarly, if a command assigns contents only to variables of level H or higher then, *a fortiori*, it assigns only to variables L or higher; thus we would have $H\ cmd \subseteq L\ cmd$. In our work, we integrated those requirements directly in the typing rules. Instead of using type coercions, we assign a fixed type to the instruction according to the more general type. For two expressions e_1 and e_2 of type τ_1 and τ_2 respectively, e_1 **op** e_2 is typed $\tau_1 \sqcup \tau_2$. For two commands c and c' typed τ and τ', the composition through sequencing or conditionals is typed $\tau \sqcap \tau'$. Another example is the typing rule CONDITIONAL_S; if we have one branch which is $L\ cmd$ and the other $H\ cmd$, the result is $L\ cmd$.

We now comment each of the typing rules. CHAN_S assigns types to channel constants, while VAR_S assigns types to variables. We assume that all integers have security type L and **skip** has type $H\ cmd$, the lowest security type of expressions and commands, respectively. OP_S assigns to the result the least upper bound of operands types.

ASSIGN-VAL_S and RECEIVE-CONTENT_S update the environment by mapping the modified variable to $(\tau \sqcup pc)\ val$, and assign the type $(\tau \sqcup pc)\ cmd$ to the command. This way if we are in a block of instructions that need to be private (i.e., $pc = H$) then the modified variable will have type H. This is typically to prevent implicit flows in **while** and **if** statements when the condition is of type H.

ASSIGN-CHAN_S and RECEIVE-NAME_S both modify a channel variable; in the first case, the type of the channel is known, in the latter, it is not, and hence the variable is given type U. In both cases, the typing is done under the condition $pc \preceq \tau$ and generates instrumentation if $pc \sqcap \tau \sqsupseteq U$. Indeed, if the source (e in one case, x_2 in the other) is of type U or H, instrumentation must be performed to insert a test that will check if the actual type of x_1 is L: if it is the case, no send can be allowed on this channel, to avoid a downward flow (and hence the channel should be marked as unsecure in the dynamic analysis). In summary, there are three cases: if $pc \npreceq \tau$, that is, $pc = H$ and $\tau = L$, the program is rejected; otherwise the program is accepted and instrumented except if $pc = L = \tau$. The case for assignation is illustrated by the second program of Fig. 1 (which uses a rule for **if** that we will explain later). In the last line, an

```
if private
then x := lowValue
end
send x to lowChan
```

```
if private
then c := publicChan
else c := privanteChan
end
send lowValue to c;
```

```
if public
then c := publicChan
else c := privateChan end
send highValue to c;
send c to lowChan;
```

Fig. 1. Treating value and channel variables in different branches of the **if** construct

information of low content is sent to c, but this cannot be allowed, as it would reveal information on our `private` condition: the sending cannot be blocked at that point; the flag must be lifted earlier, and this is when c is set to public while the context is high (because of the condition). We choose to reject such a clear flaw, which happens exactly when $pc \npreceq \tau$ (see the end of this section for a sketch on how instrumentation will handle the case $pc = U$).

For the rule RECEIVE-CONTENT_S, x_1 is given the supremum of the type of channel x_2 and pc, since it receives the content of x_2 under the context type pc. Note that RECEIVE-CONTENT_S gives to the variable x_1 a $(\tau \sqcup pc)\ val$ type while RECEIVE-NAME_S assigns a $U\ \ chan$ type.

For reasons explained earlier, SEND_S requires that the channel to which x_1 is sent must have a "maybe higher" (or equal) type than both x_1 and the context. Hence, the channel x_2, to which the content of x_1 is sent must satisfy $(\tau_1 \sqcup pc) \preceq \tau$; it assigns to the **send** instruction the type of channel x_2.

The rule CONDITIONAL_S requires to type the branches c_1 and c_2 under the type context $pc \sqcup \tau_0$. This prevents downward flows from the guard to the branches. A typical example of this situation is the first program of Fig. 1. Since x is typed under a high context (because of the `private` guard), it will obtain type H, and hence the program will be rejected, because x cannot be sent on the public channel. The typing environment produced, as stated by rule CONDITIONAL_S, is computed using the operator $\sqcup$. We now explain why $\sqcup$ is defined differently on channel variables and value variables. If Γ and Γ', the environments associated to the two branches of the **if** command, differ on a value variable, we choose to be pessimistic, and assign the supremum of the two security types. A user who prefers to obtain fewer false positive could assign type U to this variable, and leave the final decision to dynamic analysis. In the case of channel variables, we do not have the choice because the channel name can be private if it comes from a private source but its content can be public (and vice versa), as illustrated by the last program of Fig. 1. The last line should make the program rejected because the **else** branch makes c a private information, hence we could conclude that the right typing for c when typing the **if** command is H but the penultimate line **send** `highValue` **to** c; would require that c be typed as L so that the program be rejected. Hence we must type c as U, justifying the definition of $\sqcup$.

SEQUENCE_S assigns to the command the greatest lower bound of the two sequential instructions types. Note that c_1 may update the typing environment from which c_2 is typed.

Our type analysis is flow-sensitive, thus we have to analyse the **while** construct in an iterative way. To illustrate this fact, consider the program on the left-hand side of Fig. 2. In this example, it is only on the fourth iteration that

	Eval. & updates in 1st iteration	...in 2nd	...in 3rd
1. **receive**$_c$ h **from** `private`;	$h \mapsto H$		
2. $e, x_1, x_2, x_3 := 0$;	$e, x_1, x_2, x_3 \mapsto L\ val$		
3. **while** $e < 5$ **do**	$e < 5 : L\ val$		
4. **send** x_3 **to** `public`;	$[L \sqcup L \preceq L, pc \sqcup L = L]$, $\_instr \mapsto L$	ok	ok
5. $x_3 := x_2$;	$x_3 \mapsto L\ val$	-	$x_3 \mapsto H\ val$
6. $x_2 := x_1$;	$x_2 \mapsto L\ val$	$x_2 \mapsto H\ val$	-
7. $x_1 := h$;	$x_1 \mapsto H\ val$	-	-
8. $e := e + 1$	-	-	-
9. **end**			

Fig. 2. The **while** construct needs iterative analysis

we can detect a security flaw, when x_3 finally enters the body of the while with a private content and when it is sent to a public channel. The corresponding typing computation is illustrated informally on the right-hand side of the figure. The need for iterative typing analysis is treated through two rules. The first rule, LOOP1_S, is the stopping condition: typing command c under Γ does not increase the type of any variable and hence no matter how many times c is repeated, the types of variables cannot be increased. Otherwise, in LOOP2_S, if the environment is modified, we iterate using the supremum of the produced environment and the original one. We claim that this process terminates because there is a finite number of variables in c, and because the operation $\sqcup$ between the original and the produced type environment is monotonic. Indeed, variables' types are monotonically modified with respect to $\sqsubseteq$ for value variable, and with respect to the following order for channel variables: $L \rightarrow H \rightarrow U$.

We conclude this section with an example illustrating the typing of a program that needs instrumentation. Consider the program of Fig. 3. The RECEIVE-

	Evaluations & updates
1. **receive**$_{\mathbf{c}}$ h **from** `private`;	$h \mapsto H\ \mathit{val}$
2. **if** `public` **then**	$[pc \sqcup \tau_0 = L$ implies that branches context is $L]$
3. **receive**$_{\mathbf{n}}$ x **from** `private`;	$[pc = L], \quad x \mapsto U\ \mathit{chan},\ \_\mathit{instr} \mapsto U$
4. **send** h **to** x	$[pc = L,\ H \sqcup pc \preceq U], \quad \_\mathit{instr} \mapsto U$
5. **end**	

Fig. 3. A program generating unknown security values and calling for instrumentation

NAME_S rule assigns to x the security type unknown $U\ \mathit{chan}$ since we are receiving a channel name; moreover, since we receive from a private channel, instrumentation is called (to block channel x if it happens to be of low type). Another call for instrumentation is illustrated in this example, as there is an explicit flow at instruction 4 from h, whose type is H, to x, whose type is unknown.

We conclude this section by sketching out how instrumentation will be implemented and discussing when false positive arise.

False positives Let us discuss the generation of false positives, that is, cases when we reject a program that is not potentially flawed. A reject can happen from the application of one of three rules: ASSIGN-CHAN_S , RECEIVE-NAME_S and SEND_S, when $pc \npreceq \tau$, or $\tau_1 \sqcup pc \npreceq \tau$. This is only possible for the pair of values (H, L), for which $H \npreceq L$. If any of pc, τ_1 or τ is unknown (of type U), there will be no rejection, only instrumentation. According to our rules, type L can only be assigned if it is the true type of the variable, but H can be the result of a supremum taken in rule CONDITIONAL_S or LOOP_S. False positive can consequently occur from typing an **if** or **while** command whose guard prevent a bad branch to be taken; an example would be the first program of Fig. 4. If `public` is always false, the program is safe but SEND_S will reject it; this is because of the supremum taken in CONDITIONAL_S. A user who does not want

```
x := 0;                              c := highChannel
if public                            if private
then x := highValue end;             then c := lowChannel
send x to lowChannel.                end.
```

Fig. 4. False positives: uncertainty generated by **if** and assignation of a low channel in a high context

false positive at all could change the settings in order that $H\ val \sqcup L\ val = U\ val$ instead of $H\ val$. The other situation where a false positive can occur is related to assignation and reception of a channel name. Consider the second program of Fig. 4. If `private` is always false or if c is never used later on in the program, this program is harmless but still rejected. Here again, there is a way to avoid the false positive, it would be to introduce a fourth security type, B, that would mark c as unsecure and would be tested by SEND_S – but this is future work. In summary, the false positive cases that we detect happen in all other static analysis work, where they are considered reasonable, but we do suggest ways to circumvent them.

Instrumentation When the type analysis concludes that some instrumentation of the program is needed, the program will be modified in two ways. Tests will be inserted before each problematic instruction, to check if it can be safely executed; to do so, we need to update, at runtime, the type of modified variables as well as the type of the context, and hence the program must be modified in consequence. Of course, we have access to the map *TypeOf_Channel*, defined a priori. We will store the variables types in a map, *TypeOf_Var*, the security type of the context in a stack, *Ctxt*. We push the type of a condition in *Ctxt* at the beginning of a conditional or a loop and we pop a type context after each end.

In our type system there are three rules that can call for instrumentation; these "calls" happen when the variable *_instr* is assigned the value U or H. This occurs in the ASSIGN-CHAN_S rule, the RECEIVE-NAME_S rule and in the SEND_S. We will implement the type inference algorithm so as to uniquely identify statements either by labels or by the line number where they appear. The inference algorithm will save the identifier of the statement needing instrumentation, the types, variables, expressions and statements involved in the current statement. The instrumentation step will insert a test before each statement needing instrumentation. As an example, instrumenting the program of Fig. 3 will result into the one of Fig. 5. $Update(M, x, v)$ is a function that updates the map M with the association $x \mapsto v$, and *push*, *pop* and *top* are the usual functions on stacks. Note that the $\mathbf{receive_n}$ instruction marks variable x as unsecure and this information is tested when we get to the **send** instruction on line `add_4`. Hence, if a public variable was received on a private channel on line `3`, then an alert would be sent – and some action must be taken to prevent the leak of information, like aborting the program or ignoring the sending. This sketch consists in a first approach. We think it is possible to make this instrumentation lighter if we use a data flow analysis. For instance, instead of inserting the test

```
1.      receive_c h from private;
add_1.  $Update(TypeOf\_Var, h, (TypeOf\_Channel(\texttt{private}) \sqcup top(Ctxt))\ val)$;
2.      if public then
add_2.  $push(L \sqcup top(Ctxt), Ctxt)$                    ⟨because public : $L\ val$⟩
3.        receive_n x from private;
add_3.    $Update(TypeOf\_Var, x, TypeOf\_Channel(x))$;
          if $TypeOf\_Var(x) = L \wedge (TypeOf\_Var(\texttt{private}) \sqcup top(Ctxt) = H)$
          then $unsecure(x) = true$   else $unsecure(x) = false$;
add_4.  if $(TypeOf\_Var(h) \not\sqsubseteq TypeOf\_Channel(x)) \vee unsecure(x)$ then alert
4.      else send h to x end
5.      end;
add_5.  $pop(Ctxt)$
```

Fig. 5. Projected instrumentation of the program of Fig. 3

before a **send** we could insert it after a $\mathbf{receive_n}$ statement, if we know that the channel will be used in a **send** and that it could yield an illegal flow. We plan to completely specify the instrumentation in our next step for this work.

4 Type system soundness

In this section, we present the proof that our type system is sound, i.e., well-typed programs satisfy non-interference. Note that we only need to prove non-interference in case the program is typed without need of instrumentation, that is, when the type of *_instr* is L. If the type of *_instr* is U, non-interference will be guaranteed by instrumentation.

The soundness proof is inspired by the one sketched in [18]. We start by showing that our type system has the two properties listed earlier: *simple security* and *confinement*. We recall that α is used in a variable type of the form $\tau\,\alpha$ in order to replace *val* or *chan*.

Lemma 1. *(Simple security) Given a typing environment Γ, a context type pc, and an expression e, if $\Gamma, pc \vdash e : \tau\,\alpha$, then for every variable x in e, such that $\Gamma(x) = \tau'\,\alpha$, we have $\tau' \sqsubseteq \tau$.*

PROOF. By induction on the structure of e. For the base step, first assume the expression e is n or nch. Since there is no variable in e, Simple security holds trivially. Now suppose the expression e is x. By the rule VAR_S, we have $\Gamma, pc \vdash x : \tau\,\alpha$; this is obtained by $\Gamma(x) = \tau\,\alpha$ so $\tau \sqsubseteq \tau$. Simple security still holds.

For the induction step, suppose $\Gamma, pc \vdash e_1$ **op** $e_2 : (\tau_1 \sqcup \tau_2)\ val$. We have $\forall i = 1, 2$, e_i has type $\tau_i\,\alpha$; so by induction for every x_i in e_i such that $\Gamma(x_i) = \tau_i'\,\alpha$, $\tau_i' \sqsubseteq \tau_i \sqsubseteq (\tau_1 \sqcup \tau_2)$. Consequently, each of the variables type in the two expressions is lower than or equal to the final type of the expression.

Lemma 2. *(Confinement) Let Γ, Γ' be typing environments, pc be a context type and c a command. Assume that $\Gamma, pc \vdash c : \tau\ cmd, \Gamma'$ with $\Gamma'(\_instr) = L$, then for every variable x modified in c, such that $\Gamma'(x) = \tau'\,\alpha$, we have $\tau' \sqsupseteq \tau \sqcup pc$.*

We say that a variable x is modified in a command c if a value is assigned to x in c.

PROOF. By induction on the structure of c. The base step has 7 cases; the cases **skip** and **op** are trivial.
• *Case* $\Gamma, pc \vdash \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2 : \tau \sqcup pc\, cmd, \Gamma'$. According to the rule RECEIVE-CONTENT_S, channel x_2 has type $\tau\ chan$ and therefore x_1 is associated to the type $(\tau \sqcup pc)\ val$ in Γ'; thus, $\tau \sqcup pc \sqsupseteq \tau \sqcup pc$ holds. The reasoning for the rule ASSIGN-VAL_S is similar.
• *Case* $\Gamma, pc \vdash \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2 : \tau\ cmd, \Gamma'$. The only case that we have to treat is $pc = L = \tau$ because otherwise the program is either rejected or instrumented. Channel x_2 is of type $\tau\ chan$, x_1 is associated with the type U, which is greater than $\tau \sqcup pc = L$. The case ASSIGN-CHAN_S is similar.
• *Case* $\Gamma, pc \vdash \mathbf{send}\ x_1\ \mathbf{to}\ x_2 : \tau\ cmd$ with $\Gamma(x_1) = \tau_1\, \alpha$, $\Gamma(x_2) = \tau\ chan$, and $\tau_1 \sqcup pc \preceq \tau$. The **send** command does not modify any variable, but it does modify a channel. Since $\Gamma'(\_instr) = L$, we have $\tau_1 \sqcup pc \sqsubseteq \tau$, as wanted.

The induction step has two non-trivial cases; the case of **while** follows from these two.
• *Case* $\Gamma, pc \vdash c_1; c_2 : \tau\ cmd, \Gamma''$. We have $\Gamma, pc \vdash c_1 : \tau_1\ cmd, \Gamma'$ and $\Gamma', pc \vdash c_2 : \tau_2\ cmd, \Gamma''$, thus by induction, for every x_1 modified in c_1 such that $\Gamma'(x_1) = \tau_1'\ val$, we have $\tau_1' \sqsupseteq \tau_1 \sqcup pc$; similarly for c_2. Furthermore, $\tau_i \sqsupseteq (\tau_1 \sqcap \tau_2)$ and $(\tau_1 \sqcap \tau_2) = \tau$. Consequently, for every x_1 modified in c_1 such that $\Gamma'(x_1) = \tau_1'\ val$, we have $\tau_1' \sqsupseteq \tau \sqcup pc$; similarly for x_2 modified in c_2. The channel condition of confinement follows by induction.
• *Case* $\Gamma, pc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}\ : \tau_1 \sqcap \tau_2\ cmd, \Gamma_+$, where e is of type $\tau_0\ val$, $\Gamma, (pc \sqcup \tau_0) \vdash c_i : \tau_i\ cmd, \Gamma_i$, $i = 1, 2$; and let $\Gamma_+ = \Gamma_1 \sqcup \Gamma_2 \neq \bot$. If x is modified in the **if** command, then it is modified in one of the branches. We want to show that $\Gamma_+(x) \sqsupseteq (\tau_1 \sqcap \tau_2) \sqcup pc\ \alpha$. By induction, if x is modified in c_i with $\Gamma_i(x) = \tau_i'\, \alpha$, we have $\tau_i' \sqsupseteq \tau_i \sqcup pc \sqcup \tau_0$, $i = 1, 2$. We consider two cases. The first case is $\alpha = val$. By definition of $\sqcup$, $\Gamma_+(x) \sqsupseteq \Gamma_i(x)$, $i = 1, 2$, since $\tau_1' \sqcup \tau_2' \sqsupseteq \tau_i'$. Combining the two last statements on τ_i' and adding logical computations, we obtain: $\tau_1' \sqcup \tau_2' \sqsupseteq \tau_i' \sqsupseteq \tau_i \sqcup pc \sqcup \tau_0 \sqsupseteq \tau_i \sqcup pc \sqsupseteq (\tau_1 \sqcap \tau_2) \sqcup pc$, as wanted. The second case is $\alpha = chan$. The same argument applies except if $\Gamma_1(x) \neq \Gamma_2(x)$, in which case, $\Gamma_+(x) = U$; we then have to show that $(\tau_1 \sqcap \tau_2) \sqcup pc \sqsubseteq U$. If it is not the case, then either $pc = H$ or $\tau_1 = \tau_2 = H$. None of these cases is possible as $\tau_i' \sqsupseteq \tau_i \sqcup pc \sqcup \tau_0$ (established earlier) would imply that $\Gamma_1(x) = \Gamma_2(x) = H$, is a contradiction. For the channel condition of confinement : channels modified in c_i are of type $\sqsupset pc \sqcup \tau_0$ by induction and hence they are of type greater than or equal to τ, as wanted.

The following lemmas can be proved by usual induction on the structure of commands.

Lemma 3 ([18]). *Let μ and μ' be memories and c be a command. If $\langle c, \mu \rangle \rightarrow \mu'$, $x \in dom(\mu)$ and x is not modified in c, then $\mu(x) = \mu'(x)$.*

Non-interference essentially means that a variation of program input associated with a given security level does not cause variation of output of lower security level. This policy allows to manipulate and modify private data, as long as visible output does not improperly reveal information about that private data. Input data enter the program through the **receive** instructions, while output data are accessible through **send** instruction. Consequently, we define a relation $\sim_\tau$ on memories of programs that characterizes the observational power of an attacker on input and output data through communication channels.

Definition 1. *(Channel equality) Two channels* nch_1 *and* nch_2 *are equal, written* $nch_1 =_{ch} nch_2$ *if* $content(nch_1) = content(nch_2)$*, where* $content(ch)$ *is the sequence of data of channel* ch*.*

Intuitively, $nch_1 =_{ch} nch_2$ means that the channels nch_1 and nch_2 contain exactly the same information; thus, two equal channels remain equal after the same update.

Definition 2. *(τ-equivalence) Two memories* μ *and* ν *are* τ*-equivalent, written* $\mu \sim_\tau \nu$*, if* $\forall x \in dom(\mu) \cap dom(\nu) : (\Gamma(x) = \tau' \ chan \wedge \tau' \sqsubseteq \tau) \Rightarrow \mu(x) =_{ch} \nu(x)$*.*

Since we consider only one observable class of security L, we use $\sim_L$ in the following, observing that this can be generalized to more classes. The relation $\sim_L$ associates two memories that are indistinguishable to an attacker who only has public (L) access privileges.

Even though our interest is on channels, because they are the only way of communicating with a program, we need to ensure that local integer variables of type L that are equally initialized in two memories L-equivalent remain equal through program execution. Indeed, the content of a local variable can be transferred to an observable channel. This is formalized by the following lemma:

Lemma 4 ([18]). *Given two memories* μ *and* ν *that are* L*-equivalent, if for every integer variable* x *of type* L *val,* μ *and* ν *agree on the value of* x*, written* $\mu \sim_L^{val} \nu$*, then after program execution that produces the memories* μ' *and* ν'*, we have* $\mu' \sim_L^{val} \nu'$*.*

The proof is omitted due to lack of space but it is easy to see that variables can only be modified with a content coming from either equal channels (by L-equivalence), or initially equal variables.

Non-interference is formalized as follows, a variant of the definition given in [16]:

Definition 3. *(Non-interference) A program* P *satisfies non-interference if, for any memories* μ *and* ν *that are* L*-equivalent and that agree on integer variables of type* L*, the memories* μ' *and* ν' *produced by running* P *on* μ *and* ν *are also* L*-equivalent (provided that both runs terminate successfully).*

In this definition we consider that two runs of P would get exactly the same external processes writings on public channels. On the other hand, we do not treat the issue of the leak of information sensitive to termination. This issue is discussed in [1], thus we leave the non-interference definition as it was classically defined. Now let us state and prove the soundness theorem.

Theorem 1. *(Soundness theorem) Let P be a well-typed program under environment Γ, without need of instrumentation, and two L-equivalent memories μ and ν, i.e., $\mu \sim_L \nu$, that agree on integer variables of type L. If P runs successfully on both μ and ν producing the memories μ' and ν', then $\mu' \sim_L \nu'$.*

PROOF. The proof is by induction on the structure of the execution $\langle P, \mu\rangle \to \mu'$. Let $\Gamma, pc \vdash P : \tau\, cmd, \Gamma'$ with $\Gamma'(\_instr) = L$. The base step has 7 cases. The cases for **skip** and **op** are obvious and the cases of assignment are similar to the cases of $\mathbf{receive_c}$ and $\mathbf{receive_n}$:

1. Suppose $P = \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2$. The evaluation ends with $\mu' = \mu[x_1 \mapsto read(\mu(x_2))]$ and $\nu' = \nu[x_1 \mapsto read(\nu(x_2))]$. Obviously, $\mu' \sim_L \nu'$ holds as the content of channel x_2 is left unchanged.
2. Suppose $P = \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2$. The evaluation ends with $\mu' = \mu[x_1 \mapsto read(\mu(x_2))]$ and $\nu' = \nu[x_1 \mapsto read(\nu(x_2))]$. There are two cases to consider. The case $\tau \sqsupset L$ follows obviously from Confinement. The remaining case is $\tau = L$. By hypothesis, $\mu(x_2) =_{ch} \nu(x_2)$ (as μ and ν are L-equivalent) and by Definition 1, we have $read(\mu(x_2)) = read(\nu(x_2))$, thus $\mu'(x_1) =_{ch} \nu'(x_1)$. In addition, x_2 is not modified. Consequently, we have $\mu' \sim_L \nu'$.
3. Suppose $P = \mathbf{send}\ x_1\ \mathbf{to}\ x_2$ with $\Gamma(x_1) = \tau_1\,\alpha$ and $\Gamma(x_2) = \tau\ chan$. The evaluation ends with the same memories and applies the functions $update(\mu(x_2), \mu(x_1))$ and $update(\nu(x_2), \nu(x_1))$. We have two cases to consider, that respect the rule SEND_S. The first case is $\tau = L$. Since $\Gamma(\_instr)$ is not updated to U, we have $\tau_1 \sqsubseteq \tau$ and $\neg(\tau_1 = \tau = U)$; then $\tau_1 = L$. Since $\mu \sim_L^{val} \nu$ (if $\alpha =\ val$) and $\mu \sim_L \nu$ (if $\alpha =\ chan$), we have $\mu(x_1) = \nu(x_1)$. Consequently, since the channels in μ and ν are equal and updated with the same information, $\mu'(x_2) =_{ch} \nu'(x_2)$ holds and then $\mu' \sim_L \nu'$ holds. The second case is $\tau \sqsupset L$: Since $\Gamma(\_instr)$ is not updated to U, we have $\tau_1 \sqsubseteq \tau$ and $\neg(\tau_1 = \tau = U)$; then $\mu' \sim_L \nu'$ holds obviously, since no L channel is modified.

The induction step has two non-trivial cases; the case of **while** follows from these two.

1. Suppose $P = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}$, where $\Gamma, pc \vdash e : \tau_0\, cmd$, and $\Gamma, pc \sqcup \tau_0 \vdash c_i : \tau_i\, cmd, \Gamma_i$, $i = 1, 2$. If P is well typed and terminates, with typing $\Gamma, pc \vdash P : (\tau_1 \sqcap \tau_2)\, cmd, \Gamma_1 \sqcup \Gamma_2$, then c_1 and c_2 are also well typed and terminate. Assume that the execution of c_i leads to μ_i and ν_i. The first case to consider is $\tau_0 = L$. By L-equivalence, the evaluations of e under μ and ν are equal, and hence the same branch c_i is taken for both memories. By induction, μ_i and ν_i are also L-equivalent, and the case is proven. The second case is $\tau_0 \sqsupset L$. By confinement, since c_1 and c_2 are typed under a context of type strictly greater than L, no variable of type L is modified by their execution, and hence $\mu_1 \sim_L \mu_2 \sim_L \nu_1 \sim_L \nu_2$, and the case is proven.
2. Suppose $P = c_1; c_2$. By induction, the execution of c_1 on L-equivalent memories μ and ν leads to $\mu_1 \sim_L \nu_1$ By induction again, executing c_2 on μ_1 and ν_1 results in two L-equivalent memories.

5 Related work

Securing flow information has been widely studied since the late seventies. Denning and Denning [6] introduced secure information-flow by static analysis, based on control and data flow analysis. They defined implicit and explicit flow and devised static analysis and dynamic analysis based on lightly instrumenting the target program. An important approach based on type analysis has attracted many researchers. Many static approaches have been devised based on type systems, they addressed languages with different levels of expressivity.

Volpano and Smith [18] devised a type based analysis for an imperative language. Pottier and Simonet treat in [11] the functional language ML, supporting references, exceptions and polymorphism. In [9] Myers statically enforces information flow policy in JFlow, an extension of Java that adds level security annotations to variables, making information flow checking more precise and flexible. JFlow supports objects, subclassing, dynamic type tests, access control, and exceptions. Banerjee and Naumann devised a type based analysis in[2] that ensures secure flow information. They treat the object-oriented language Java. Barthe and al. in [3] and Terauchi and Aiken in [17] investigated logical-formulation of non-interference, enabling the use of theorem-proving or model-checking based techniques. In [14] Sabelfeld and Sands extend type system approaches to support higher order functions and non determinism. We mention some studies on other notions of non-interference, possibilistic and probabilistic non-interference, which treat information flow security for concurrent programs, see [4, 13, 15].

Purely static approaches suffer from a large number of false positives, leading to reject programs that may be flow information secure. An analysis has been proposed to take into account data flow information [7], this approach is called flow sensitive type approach. Combining static and dynamic approaches has been proposed by Russo and Sabelfeld in [12], where the authors prove that this approach reject less safe programs. Their approach is based on calling static analysis during execution.

Similarly to [7] and [12] our approach is flow sensitive, it differs in that we distinguish between variables in live memory and channels. We consider that programs interact with their environment through information flow via channels and that the security level is associated to an information according to its source, the channels from which this value is computed. We propose a three valued typing analysis aiming at the distinction between the cases where the analysis is uncertain due to lack of information and the cases where it is certain. Rather than calling static analysis during execution, our approach prepare for a light instrumentation of the code only when there is a need for it.

6 Conclusion

Ensuring secure information flow within sensitive systems has been studied extensively. In general, the key idea in type-based approaches is that if a program is well typed according to its typing rules, then it is secure according to given security properties.

We define a sound type system that captures lack of information in a program at compile-time. Our type system is flow-sensitive, variables are assigned the security levels of the values they store. We make a clear distinction between local variables and channels through which the program communicates. This makes our analysis more realistic.

Our main contribution is the handling of a three-valued security typing. The program is considered well typed, ill typed or uncertain. In the first case, the program can safely be executed, in the second case the program is rejected and need modifications, while in the third case instrumentation is to be used in order to guarantee the satisfaction of non-interference. This approach allows to eliminate false positives due to conservative static analysis approximations and to introduce run-time overhead only when it is necessary. We obtain fewer false positives than purely static approaches because we send some usually rejected programs to instrumentation.

Future work includes the extension of our type analysis to make it a complete security hybrid analysis, using data-flow analysis and code instrumentation. In short, we will insert tests before the instructions that were detected as possibly faulty during the type analysis.

Acknowledgements The authors wish to thank the first anonymous referee whose arguments were really helpful and valuable in producing the final version of this paper.

References

1. Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the European Symposium on Research in Computer Security: Computer Security, ESORICS'08*, 2008.
2. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the IEEE Computer Security Foundations Workshop*, 2002.
3. Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2004.
4. Gilles Barthe and Leonor Prensa Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15:647–689, 2007.
5. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, May 1976.
6. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, July 1977.
7. Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 2006.
8. Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
9. Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.

10. Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, July 2006.
11. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25:117–158, January 2003.
12. Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.
13. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2000.
14. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
15. Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.
16. Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, volume 27, pages 291–307. Springer, 2007.
17. Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Proceedings of the International Symposium on Static Analysis*, 2005.
18. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.