Enforcing information flow by combining static and dynamic analysis

Andrew Bedford, Josée Desharnais,
Théophane G. Godonou, and Nadia Tawbi

Université Laval, Qubec, Canada
{andrew.bedford.1,theophane-gloria.godonou.1}@ulaval.ca
{josee.desharnais,nadia.tawbi}@ift.ulaval.ca

Abstract. This paper presents an approach to enforce information flow policies using a multi-valued type-based analysis followed by an instrumentation when needed. The target is a core imperative language. Our approach aims at reducing false positives generated by static analysis, and at reducing execution overhead by instrumenting only when needed. 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 the possibly faulty commands, they are identified and annotated for the second step of the analysis; the positive and negative responses are treated as is usually done. This work is a hybrid security enforcement mechanism: the *maybe-secure* points of the program detected by our type based analysis are instrumented with dynamic tests. The basic type based analysis has been reported in [6], this paper presents the modification of the type system and the newly presented instrumentation step. The novelty of our approach is the handling of four 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. Thus the analysis is flow-sensitive.

1 Introduction

In today's world, we depend on information systems in many aspects of our lives. Those systems are interconnected, rely on mobile components and are more and more complex. Security issues in this context are a major concern, especially when it comes to securing information flow. How can we be sure that a program using a credit card number will not leak this information to an unauthorized person? Or that one that verifies a secret password to authenticate a user will not write it in a file with public access? Those are examples of information flow breaches in a program that should be controlled. 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 multi-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 two values, *Unknown* and *Blocked*. The former was introduced in [6] and captures the possibility that we may not know, before execution, whether some 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. In this paper, we add a fourth security type, *Blocked*, which is used to tag a public channel variable that must not receive any information, even public, because its value (the name of the channel) depends on private information. As long as no information is sent over such a channel, the program is considered secure.

The program on the left of Figure 1 shows how the blocking type results in fewer false positive alarms. The figure also exhibit our analysis of the program (which we will explain later) as well as the output given by our implementation. The identifiers *privateChannel*, *publicChannel*, *highValue* and *lowValue* in all the examples are predefined constants. The security types L, H, U, B represent *Low*, *High*, *Unknown* and *Blocked*, respectively, *pc* is the security type of the context, and $\_instr = L$ to tell that there is no need for instrumentation. The first four lines of the program would be rejected by other analyses, including [6], because channel c is assigned a *Low* channel in the **then** branch, which depends on a private condition, *highValue*. In our work, c is just marked as blocked ("$c \mapsto B chan$") when it is assigned a public channel in a private context. However, in the last line, an information of low content is sent to c, which cannot be allowed, as it would reveal information on our confidential condition *highValue*. It is because of this last line that the program is rejected by our analysis: without it, c is just typed as B.

Input to analyzer	Inference analysis		
	Environment	*pc*	*i*
		$pc_{if} = H$	2
if *highValue*	$G(2) = [\_instr \mapsto L, c \mapsto B\ chan]$	H	3
then $c :=$ *publicChannel*	$G(3) = [\_instr \mapsto L, c \mapsto H\ chan]$	H	4
else $c :=$ *privateChannel*	$G(1) = [\_instr \mapsto L, c \mapsto B\ chan]$	H	4
end;	**fail** since $c \mapsto B\ chan$		
send *lowValue* **to** c			
Output : Error (Send) : Cannot send *lowValue* to channel c because it is blocked.			

Fig. 1. Analysis of a program where an implicit flow may lead to a leak of information

The goal of our 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 interact with an external environment through communication *channels*, i.e., objects through which a program can exchange information with users (printing screen, file, network, etc.). In contrast with the work of Volpano et al. [21], 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 [10, 17]. 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 [4]: *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. Thus, the security requirements are:

- explicit flows from a variable to a channel of lower security are forbidden;
- implicit flows where the guard contains a variable of higher security than the variables assigned are forbidden.

Our static analysis is based on the typing system of [6]; our contributions are an improvement of the type system to allow fewer false positive, by the introduction of the blocked type, and the instrumentation algorithm that we have developed and implemented [3].

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 inference algorithm is presented in Section 4. The instrumentation algorithm is presented in Section 5. Section 6 is dedicated to related work. We conclude in Section 7.

2 Programming language

We illustrate our approach on a simple imperative programming language, introduced in [6], a variant of the one in [19], which was adapted to deal with the communication via channels.

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:

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

Values are integers (we use zero for false and nonzero for true), or channel names. The symbol **op** stands for arithmetic or logic binary operators on integers 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 [15, 12]. We consider that external processes can only read and write to public channels. The instructions related to accessing channels deserve further explanations.

The instruction $\mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2$ 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. The instruction $\mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2$ 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. The instruction $\mathbf{send}\ x_1\ \mathbf{to}\ x_2$ is 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 programs follows a commonly used operational semantics [6]; we present a few rules in Table 1. An instruction p is executed under a memory map $\mu : \mathcal{V}ar \to \mathbb{N} \cup \mathcal{C}$. Hence the semantics specifies how *configurations* $\langle p, \mu \rangle$ evolve, either to a value, 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

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

(ASSIGN)	$\dfrac{\langle e, \mu\rangle \rightarrow_e v}{\langle \mathbf{x} := \mathbf{e}, \mu\rangle \rightarrow \mu[x \mapsto v]}$
(RECEIVE-CONTENT)	$\dfrac{x_2 \in \mathrm{dom}(\mu) \qquad read(\mu(x_2)) = n}{\langle \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto n]}$
(RECEIVE-NAME)	$\dfrac{x_2 \in \mathrm{dom}(\mu) \qquad read(\mu(x_2)) = nch}{\langle \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto nch]}$
(SEND)	$\dfrac{x_1 \in \mathrm{dom}(\mu)}{\langle \mathbf{send}\ x_1\ \mathbf{to}\ x_2, \mu\rangle \rightarrow \mu, update(\mu(x_2), \mu(x_1))}$
(CONDITIONAL)	$\dfrac{\langle e, \mu\rangle \rightarrow_e n \qquad n \neq 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \langle c_1, \mu\rangle}$
	$\dfrac{\langle e, \mu\rangle \rightarrow_e n \qquad n = 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \langle c_2, \mu\rangle}$

Table 1. A few rules of the structural operational semantics

by $\rightarrow_e$. Finally, $\langle c, \mu\rangle$ leads to a memory produced by the execution of command c on memory μ; this transition is designated by $\rightarrow$.

We explain the rules that manipulate channels. The instructions $\mathbf{receive_c}\ x_1$ **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 kinds of receive.

The instruction **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**.

3 Security type system

We now present the security type system that we use to check whether a program of the language described above, either satisfies non-interference, may satisfy it or does not satisfy it. It is an improvement of the one introduced in [6]: we add a security level, B, to tag a channel that should be blocked.

The security types are defined as follows:

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

We consider a set of four security levels $SL = \{L, U, H, B\}$. This set is extended to a lattice $(SL, \sqsubseteq)$ using the following order: $L \sqsubseteq U \sqsubseteq H \sqsubseteq B$ (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 phrase types in the trivial way, and assume this returns $\bot$ when applied to phrases of different types, e.g., $H\ chan \sqcup H\ val = \bot$.

When typing a program, security types are assigned to variables, channels and commands, hence phrase types – and to the context of execution. The meaning of types is as follows. A variable of type $\tau\ val$ has a content of security type τ; a channel of type $\tau\ chan$ 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 $\tau\ cmd$ 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 $L\ chan$. Type B will only be assigned to channels, to indicate that they were of type $L\ chan$ but must be blocked, to avoid an implicit flow. The context type pc represents the type of the surrounding conditionals and helps in indicating implicit flows.

Our type system satisfies two natural properties: *simple security*, applying to expressions and *confinement*, applying to commands [19]. *Simple security* says that an expression e of type $\tau\ val$ or $\tau\ chan$ 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 $\tau\ cmd$ 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. The complete soundness proof of this algorithm is similar to the one presented in [7].

Our typing rules are shown in Table 2. They are the same as in [6] except for the three rules that deal with channels. 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 $\tau\ val$ or $\tau\ chan$ that represents 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 higher 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.

We use, as in [6], a special variable *_instr*, whose type (maintained in the typing environment map according to the typing rules) tells whether or not the program needs instrumentation. The initial value of *_instr* is L; if the inference algorithm detects a need for instrumentation, its value is changed to U, H or

(CHAN_S) $\dfrac{\mathit{TypeOf\_Channel}(nch) = \tau}{\Gamma, pc \vdash nch : \tau\ chan}$ (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) $\dfrac{\Gamma, pc \vdash e : \tau\ chan}{\Gamma, pc \vdash x := e : \tau\ cmd, \Gamma \sqcup [\_instr \mapsto HL^L_L(pc, \tau)] \dagger [x \mapsto HL^B_\tau(pc, \tau)chan]}$

(RECEIVE-CONTENT_S) $\dfrac{\Gamma(x_2) = \tau\ chan}{\Gamma, pc \vdash \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2 : (\tau \sqcup pc)\ cmd, \Gamma \dagger [x_1 \mapsto (\tau \sqcup pc)val]}$

(RECEIVE-NAME_S) $\dfrac{\Gamma(x_2) = \tau\ chan}{\Gamma, pc \vdash \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2 : \tau\ cmd, \Gamma \sqcup [\_instr \mapsto HL^L_\tau(pc, \tau)] \dagger [x_1 \mapsto HL^B_{U \sqcup \tau}(pc, \tau)chan]}$

(SEND_S) $\dfrac{\Gamma(x_1) = \tau_1\,\alpha \quad \Gamma(x_2) = \tau\ chan \qquad \neg((\tau_1 \sqcup pc) = H \wedge \tau = L) \qquad \tau \neq B}{\Gamma, pc \vdash \mathbf{send}\ x_1\ \mathbf{to}\ x_2 : \tau\ \ cmd, \Gamma \sqcup [\_instr \mapsto HL^U_L(\tau_1 \sqcup pc, \tau)]}$

(CONDITIONAL_S) $\dfrac{\Gamma, pc \vdash e : \tau_0 val \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_1 : \tau_1\ cmd, \Gamma' \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_2 : \tau_2\ cmd, \Gamma'' \qquad \Gamma' \sqcup \Gamma'' \sqsupset \bot}{\Gamma, pc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end} : (\tau_1 \sqcap \tau_2)\ cmd, \Gamma' \sqcup \Gamma''}$

(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) $\dfrac{\Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ \ cmd, \Gamma' \quad \Gamma \neq \Gamma \sqcup \Gamma' \sqsupset \bot \qquad \Gamma, pc \vdash e : \tau_0 val \quad \Gamma \sqcup \Gamma', (pc \sqcup \tau_0) \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ cmd, \Gamma''}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ cmd, \ \Gamma''}$

(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

B, depending on the rule applied, most of the time depending on the type of a channel. When it is updated, the supremum operator is always involved to make sure that the need for instrumentation is recorded until the end.

We need to define three operators, two of which on typing environments: $\Gamma \dagger [x \mapsto \rho]$ and $\Gamma \sqcup \Gamma'$. 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 in particular, we need 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 and none of them is blocked.

Definition 1. *The supremum of two environments is given as* $dom(\Gamma \sqcup \Gamma') = dom(\Gamma) \cup dom(\Gamma')$*, and*

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

Note that $\Gamma \sqcup \Gamma'(x)$ can return $\bot$ 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).

In the three rules that modify a channel, ASSIGN-CHAN_S, RECEIVE-NAME_S et SEND_S, the following operator is also used.

Definition 2. *The function HL computes the security level of _instr and channel variables in the three typing rules where a channel is modified.*

$$HL^{\psi}_{\nu}(pc, \tau) = \begin{cases} \psi \text{ if } (pc, \tau) = (H, L) \\ U \text{ if } (pc, \tau) \in \{(U, L), (U, U), (H, U)\} \\ \nu \text{ otherwise.} \end{cases} \quad \text{where } \psi, \nu, pc, \tau \in SL.$$

The notation *HL* refers to a downward flow "H to L" because this (handy and maybe tricky) function encodes (with ψ and ν), in particular, how such a flow from pc to τ should be handled. When it is clear that there is a downward flow, from H to L, then *HL* returns type ψ. When we are considering the security type of a channel variable, ψ is either U or B. Such a flow may not lead to a rejection of the program, nor to an instrumentation: when a variable is blocked, there is no need to instrument. For other flows, the analysis distinguishes between safe flows and uncertain ones. For example, flows from U to H are secure, no matter what the types of uncertain variables actually are at runtime (L or H). In these cases, $HL^{\psi}_{\nu}(pc, \tau)$ returns ν. However, depending on the actual type of the U variable at runtime, a flow U to L, from U to U or from H 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. Hence, in these cases, *HL* will return U.

In related work, there are *subtyping judgements* of the form $\rho_1 \subseteq \rho_2$ or $\rho_1 \leq \rho_2$ [19, 21]. 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 with $\tau_1 \sqcup \tau_2$. For two commands c and c' typed τ and τ', the composition through sequencing or conditionals is typed with $\tau \sqcap \tau'$.

We now comment the typing rules that are modified with respect to [6]. ASSIGN-CHAN_S and RECEIVE-NAME_S both modify a channel variable and they make use of the function HL^{ψ}_{ν}. The usual condition for the modification of a

channel would be to avoid downward flow by imposing $pc \sqsubseteq \tau$ or, as in [6], $pc \preceq \tau$; the latter is a weakening of the former, that returns false only if $pc = H$ and $\tau = L$. In this paper, we chose to only reject a program if an unauthorized **send** is performed. If we detect an implicit flow in ASSIGN-CHAN_S or RECEIVE-NAME_S, that is, $pc = H$ and $\tau = L$, we rather block the assigned channel (by $\psi = B$ in HL^B_-), as in the program of Figure 1; if the channel is never used, a false positive has been avoided. If the channel is blocked, there is no need for instrumentation, hence $\psi = L$ in HL^L_- for both rules. In RECEIVE-NAME_S, we must call instrumentation when τ is U or H to prevent a downward flow from x_2 to x_1. In that case, the channel variable obtains security type $U \sqcup \tau$ because its type is unknown: we could receive the name of a private channel on a public one (but could not read on in). In ASSIGN-CHAN_S, this type is τ, the type of the assigned expression.

The rule for SEND_S states that the typing fails in two situations where the leak of information is clear: either the channel to which x_1 is sent is blocked ($\tau = B$), or it is of type L and either the context or the variable sent has type H ($(\tau_1 \sqcup pc) = H$). An example where $\tau = B$ was just discussed above. If the typing does not fail, the instrumentation will be called in each case where there is a possibility, at runtime, that $\tau_1 \sqcup pc$ be H while the channel has type L; those are the cases $(\tau_1 \sqcup pc, \tau) \in \{((U, L), (U, U), (H, U)\}$. The "$\psi$ branch" in the definition of HL^{ψ}_- is useless, as it is a case where the typing rejects the program.

The rule CONDITIONAL_S requires to type the branches c_1 and c_2 under the type context $pc \sqcup \tau_0$, to prevent downward flows from the guard to the branches.

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: different unblocked channels must obtain the type $U\,chan$. The program on the left of Figure 3 illustrates why. The last line of the program would require that c be typed as $L\,chan$ so that the program be rejected. However, since the **else** branch makes c a private information, a command like **send** c **to** *publicChannel* should also be rejected, and hence in this case we would like that c had been typed $H\,chan$. Hence we must type c as $U\,chan$, justifying the definition of $\sqcup$. Interestingly, this also shows that in our setting, the uncertain typing is necessary.

We conclude this section by discussing occurrences of false positive alarms. A rejection can only happen from the application of the rule SEND_S: either the channel to which x_1 is sent is blocked, or it is of type L and the context, or the variable sent, is of type H. 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 always prevent a "bad" branch to be taken. This is unavoidable in static analysis, unless we want to instrument any uncertainty arising from the values of guards. Nevertheless, our inference typing

rules prevent more false positives than previous work through the blocking of channels and the unknown types U.

4 Inference Algorithm

The inference algorithm implements the specification given by the type system together with some refinements we adopted in order to prepare for the instrumentation step. The refinements consist in keeping track of the command line number and of the generated environment for this command. Although it may seem overloading, this strategy lightens the dynamic step since it avoids type inference computation whenever it is already done. The algorithm is implemented as the function `Infer` which is applied to the current typing environment, $g_e : \mathcal{Var} \to \{L, H, U, B\}$, a number identifying the current command to be analyzed, the command line i, the security level of the current context, pc, and the actual command to be analyzed, c . Along the way, `Infer` returns a typing environment representing the environment valid after the execution of the command c and an integer representing the number identifying the next command to be analyzed. `Infer` updates $G : int \to (\mathcal{Var} \to \{L, H, U, B\})$ as a side effect; G associates to each command number a typing environment valid after its execution. Recall that the environment associates to a specific variable $\_instr$ a security level. After the application of the inference algorithm, if the program is not rejected and the resulting environment associates U, H or B to $\_instr$ then the program needs instrumentation, otherwise it is safe w.r.t. non-interference.

To analyze a program P, `Infer` is invoked with $g_e = [\_instr \mapsto L]$, $i = 0$, $pc = L$ and $c = P$. The inference algorithm uses a set of utility functions that implement some operators, functions and definitions appearing in the typing system. Their meaning and their implementation are straightforward. Here is the list of these functions. The set $SecType$ stands for $\{\tau \quad v : \tau \in \{L, U, H, B\}, v \in \{val, chan\}\}$, t and t_i ranges over $SecType$, and g_i ranges over Env, `lessOrEqual` implements $\sqsubseteq$, `inf` and `sup` implement respectively the infimum and the supremum of two security levels. `supEnv` implements the supremum of two environments, as in Definition 1. `infV` : $SecType \times SecType \to SecType \cup \{\bot_T\}$ returns $\bot_T$ if the two security types do not have the same nature. If the nature is the same, then it returns a security type where the security level is the infimum of the two security types given as argument, `supV` : $SecType \times SecType \to SecType \cup \{\bot_T\}$ behaves the same way as `infV` except that it returns a security type where the security level is the supremum of the two security types given as argument, `incBottomEnv` : $Env \to bool$ returns true if at least one variable is associated to $\bot_T$ in its parameter, `updateEnv` : $Env \times \mathcal{Var} \times SecType \to Env$ implements $\Gamma \dagger [x \mapsto \rho]$, `eqEnv` : $Env \times Env \to bool$ checks if two environments are equal. It returns true if the two environments have the same domain and all their variables have the same security type. It returns false otherwise, `evalN` : $SecType \to \{val, chan\}$, extracts the nature of a security type (val or $chan$), `evalT` : $SecType \to \{L, U, H, B\}$, extracts the level of a security type, `inferE` : $Env \times Exp \to SecType$ returns the highest security type of the variables present in the expression to which it is applied, and `HL` : $\{L, U, H, B\}^4 \to \{L, U, H, B\}$ implements the function HL as in Definition 2.

$\texttt{Infer}: Env \times int \times Sec \times\ cmd \rightarrow Env \times int$

$\texttt{Infer}(g_e, i, pc, c) =$
  case c of
    skip : $G(i) = g_e$
      return $(G(i), i+1)$
    x := **e** :
      $\tau = \texttt{evalT}(\texttt{inferE}(g_e, e))$
      case $\texttt{evalN}(\texttt{inferE}(g_e, e))$ of
        val: $G(i) = \texttt{updateEnv}(g_e, x, \texttt{sup}(pc, \tau)\ \ val)$
          return $(G(i), i+1)$
        $chan$: $\_instr_t = \texttt{HL}(L, L, pc, \tau)$
          $x_t = \texttt{HL}(B, \tau, pc, \tau)$
          $\_instr_{t2} = g_e(\_instr)$
          $G(i) = \texttt{updateEnv}(\texttt{updateEnv}(g_e, \_instr, sup(\_instr_t, \_instr_{t2})), x, x_t\ \ chan)$
          return $(G(i), i+1)$
    receive$_\mathbf{c}$ x_1 **from** x_2 :
      $\tau = \texttt{evalT}(g_e(x_2))$
      $G(i) = \texttt{updateEnv}(g_e, x_1, \texttt{sup}(pc, \tau)\ \ val)$
        return $(G(i), i+1)$
    receive$_\mathbf{n}$ x_1 **from** x_2 :
      $\tau = \texttt{evalT}(g_e(x_2))$
      $\_instr_t = \texttt{HL}(L, \tau, pc, \tau)$
      $\_instr_{t2} = g_e(\_instr)$
      $x1_t = \texttt{HL}(B, \texttt{sup}(U, \tau), pc, \tau)$
      $G(i) = \texttt{updateEnv}(\texttt{updateEnv}(g_e, \_instr, \texttt{sup}(\_instr_t, \_instr_{t2})), x_1, x1_t\ \ chan)$
      return $(G(i), i+1)$
    send x_1 **to** x_2 :
      $\tau_1 = \texttt{evalT}(g_e(x_1))$
      $\tau = \texttt{evalT}(g_e(x_2))$
      $\_instr_t = \texttt{HL}(U, L, \texttt{sup}(\tau_1, pc), \tau)$
      $\_instr_{t2} = g_e(\_instr)$
      if$((\tau \neq B)\ and\ \neg(\texttt{sup}(\tau_1, pc) = H\ and\ \tau = L)))$
        then $G(i) = \texttt{updateEnv}(g_e, \_instr, \texttt{sup}(\_instr_t, \_instr_{t2}))$
        else **fail**
      return $(G(i), i+1)$
    c$_1$; **c$_2$** :
      $(g_1, j) = \texttt{Infer}(g_e, i, pc, c_1)$
      $(g_2, k) = \texttt{Infer}(g_1, j, pc, c_2)$
      return (g_2, k)
    if e **then** c_1 **else** c_2 **end**:
      $t = \texttt{evalT}(\texttt{inferE}(g_e, e))$
      $pc_{if} = \texttt{sup}(pc, t)$
      $(g_1, j) = \texttt{Infer}(g_e, i+1, pc_{if}, c_1)$
      $(g_2, k) = \texttt{Infer}(g_e, j, pc_{if}, c_2)$
      if$(\neg\texttt{incBottomEnv}(\texttt{supEnv}(g_1, g_2)))$ then $G(i) = \texttt{supEnv}(g_1, g_2)$
        else **fail**
      return $(G(i), k)$
    while e **do** c **end**:
      $t = \texttt{evalT}(\ \texttt{inferE}(g_e, e))$
      $pc_{while} = \texttt{sup}(pc, t)$
      $(g_{e'}, j) = \texttt{Infer}(g_e, i+1, pc_{while}, c)$
      if $(\texttt{eqEnv}(g_e, \texttt{supEnv}(g_e, g_{e'}))$ and $(\neg\texttt{incBottomEnv}(\texttt{supEnv}(g_e, g_{e'}))))$
        then $g_{res} = \texttt{supEnv}(g_e, g_{e'})$
        else if $(\neg\texttt{eqEnv}(g_e, \texttt{supEnv}(g_e, g_{e'}))$ and $(\neg\texttt{incBottomEnv}(\texttt{supEnv}(g_e, g_{e'}))))$
          then $(g_{res}, j) = \texttt{Infer}(\texttt{supEnv}(g_e, g_{e'}), i, pc_{while},$ **while** e **do** c **end**$)$
          else **fail**
      $G(i) = \texttt{supEnv}(G(i), gres)$
      return (g_{res}, j)

Table 3. Inference Algorithm

The inference algorithm `Infer` is presented in Table 3. Some examples of its output are presented in the following section.

Correctness proof To guarantee the correction of the algorithm we have to prove its soundness and its completeness w.r.t. the type system.

Theorem 1. *(Correctness of* `Infer` *algorithm) Let P be a program in our target language, G a typing environment, and pc a security level,* $\texttt{Infer}(G, 0, pc, P) = (G', j) \iff G, pc \vdash P : rho, G'$*, and* `Infer` *rejects P if and only if the typing system leads to a similar conclusion.*

5 Instrumentation

Our instrumentation is based on the inference algorithm. It is a new contribution w.r.t. [6]. It inserts commands in the program so that, during execution, the program will update the security level of variables which were unknown (U) statically. Each instruction is treated with its corresponding line number and its context security level.Instructions may be inserted to prevent unsecure executions. The instrumentation algorithm is shown in Table 4; it is given a command *cmd* to instrument and the number of this command. The algorithm updates IC : *String* as a side effect, which is the instrumented program; it uses the matrix of typing environments G produced by the inference algorithm, which is a global variable. *G(i)* refers to the typing environment of instruction i, and hence *G(i)(x)* is the security type of variable x at instruction i.

Commands are inserted so that the instrumented program will keep a table *g_M* of the security levels of variables, picking the already known types in G. This table is also a global variable. *g_M* offers two advantages, it keeps track of the most recent values of the variables. No further analysis is necessary to find which instruction was the last to modify the variables. It is also easier to read the value from a table than from the matrix G. The usefulness of *g_M* can be shown with the following example.

receive$_{\mathbf{n}}$ c **from** *publicChannel*;
receive$_{\mathbf{c}}$ a **from** *publicChannel*;
if ($a \mod 2 \neq 0$) **then**
  receive$_{\mathbf{c}}$ a **from** c
end;
send a **to** *publicChannel*

The inference algorithm determines after the first instruction that the type of c is U *chan*. Variable a, before the **if** command, has the type L *val*. The static analysis will conclude that the type of a after executing the **if** command is U *val*. If the instrumented program updates the variables immediately in G, the type of a would be H *val*. The following **send** would be considered unsecure no matter what the dynamic value of a is. Our instrumentation will insert instructions that put in *g_M* the last type of a when it was read. So depending on whether the execution of the instrumented program enters the **then** branch or not, a will take either the security level of c, or it will keep the security level of *publicChannel*. This will allow the instrumented program to be rejected during execution only if it is actually unsecure.

A set of utility functions are predefined in the target language and used by the instrumented programs. Function TypeOf_Channel serves as a register for the

```
Instrument: cmd * int → int

Instrument(c, i) =     case c of
     skip : IC ∧ " skip; "
      return (i + 1)
      x := e :
         τ = evalT(inferE(G(i), e))
         case evalN(inferE(G(i), e)) of
           val: IC = IC∧ "x := e ; "
                if (τ = U) then
                     IC = IC∧"updateEnv(G(i), x, sup(evalT(TypeOf_Expression(e)), top(pc))val); "
                end ;
                IC = IC∧ " updateEnv(g_M, x, G(i)(x)) ; "
             return (i + 1)
           chan: IC = IC∧ "x := e ; "
                 if  (τ = U) then
                       IC = IC∧" updateEnv(G(i), x, TypeOf_Channel(e)); "
                 end
                 IC = IC∧ " updateEnv(g_M, x, G(i)(x)); "
             return (i + 1)
      receive_c x1 from x2 :
         IC = IC ∧ "receive_c x1 from x2; "
         if  (G(i)(x1) = Uval)  then
              IC = IC∧ " updateEnv(G(i), x1, sup(evalT(TypeOf_Expression(x2)), top(pc))val) ; "
         end
         IC = IC∧ "updateEnv(g_M, x1, G(i)(x1)) ; "
         return (i + 1)
      receive_n x1 from  x2 :
         IC = IC∧ "receive_n x1 from x2 ";
         if  (G(i)(x2) != L chan)
           then  IC = IC∧ " if TypeOf_Channel(x1) = L chan and TypeOf_Channel(x2) = H chan
               then updateEnv(G(i), x1, B chan)
               else updateEnv(G(i), x1, TypeOf_Channel(x1))
              end "
            else IC = IC∧ "updateEnv(G(i), x1, TypeOf_Channel(x1))"
            end
         IC = IC∧ " updateEnv(g_M, x1, G(i)(x1)); "
      return (i + 1)
      c1; c2 :
         j = Instrument(c1, i); k = Instrument(c2, j)
         return k
      send x1 to x2 :
            IC = IC∧ " tau = TypeOf_Expression(x2) ; tau_l = TypeOf_Expression(x1);
            if(((tau = L chan) and (sup(evalT(tau_l), top(pc)) = H)) or (tau = B chan))
            then fail else send x1 to x2 end; "
      return (i + 1)
      if e then c1 else c2 end :
         IC = IC∧ "push(sup(top(pc), evalT(TypeOf_Expression(e))), pc) ;
         if e then "
         j = Instrument(c1, i + 1)
        IC = IC∧ "else "
         k = Instrument(c2, j)
         IC = IC∧ "end;
         pop (pc); "
      return k
      while e to c end:
      IC = IC∧
         "push(sup(top(pc), evalT(TypeOf_Expression(e))), pc);
         while e do "
      j = Instrument(c, i + 1)
      IC = IC∧ "end ;
         pop (pc) ; "
      return j
```

Table 4. Instrumentation algorithm.

constant channels defined prior to the execution. Function TypeOf_Expression returns the actual type of an expression: it uses the information of g_M for variables, TypeOf_Channel for actual channels and takes the supV of these values when the expression is e_1**op** e_2. TypeOf_Expression and TypeOf_Channel are commands executed by the instrumented program. To prevent implicit flows, commands are inserted so that the instrumented program will keep a stack of contexts *pc*. Each time the execution branches on an expression, whether it is in an **if** or a **while** command, the context is pushed onto the stack. The context is the supremum of the type of expression e and the context in which the actual command is executed. The last context is popped from the stack everytime the execution of a branching command finishes. The stack *pc* is initially empty and the result of reading an empty stack is always L. The functions push and pop are used to manipulate the stack of contexts during the execution of the instrumented program. The remaining functions are an implementation of their counter part in the algorithm `Infer`.

The analysis and the instrumentation has been implemented. The interested reader can find a link to the implemented code in [3]. The implementation is divided into two parts : an analyzer and an interface. The analyzer is written in OCaml. It uses OCamllex and OCamlyacc to generate the lexer and parser. In order to maximize the portability of our application, we use OCaml-Java to compile our OCaml code into bytecode so that it may run on a Java Virtual Machine. As for the interface, it is written in Java and uses the standard Swing library. If an error is detected while analyzing, whether it is a lexical, syntactic, semantic or flow error, the analyzer stops and displays a message explaining the cause of the error. If the analyzer infers that the code needs to be instrumented, it automatically generates and displays the instrumented code. If no error occurs and there is no need for instrumentation, then a message of correctness is displayed.

Examples A few examples of the whole approach are presented in the following figures. The figures show the returned environment G, the returned command number i as well as the input *pc*, the security level of the context. Recall that the identifiers *privateChannel*, *publicChannel*, *highValue* and *lowValue* are predefined constants. The result of the analysis, including instrumentation when necessary, is shown in the lower part of the figures.

The program of Figure 2 is rejected. The security level of the value variable x is H because its value is assigned inside the context of *highValue*, which is of type H. There is an attempt to send x on a public channel, which make the program be rejected.

Input to analyzer	Inference analysis		
	Environment	*pc*	*i*
if *highValue*		$pc_{if} = H$	2
then x := *lowValue*	$G(2) = [\_instr \mapsto L, x \mapsto H\ val]$	H	3
else skip	$G(3) = [\_instr \mapsto L]$	H	4
end;	$G(1) = [\_instr \mapsto L, x \mapsto H\ val]$	H	4
send x **to** *publicChannel*	**fail** since $H \not\sqsubseteq L$		
Output : Error (Send) : Cannot send x (H) to publicChannel (L).			

Fig. 2. Implicit flow

The program in Figure 3 is similar to the one in Figure 1 except that the context in which c is defined is now L instead of H. For this reason, it is not necessary to block channel c. Since c can either be a public or private channel (depending on the value of *lowValue*), it is marked as unknown. A call for instrumentation results from the first send, to ensure that *highValue* is only sent to a private channel.

<table>
<tr><th>Input to analyzer</th><th colspan="4">Inference analysis</th></tr>
<tr><td rowspan="6">if $lowValue$
  then $c := publicChannel$
  else $c := privateChannel$
end;
send $highValue$ to c</td><td>Environment</td><td>pc</td><td>i</td></tr>
<tr><td>$G(1) = [\_instr \mapsto L]$</td><td>$pc_{if} = L$</td><td>2</td></tr>
<tr><td>$G(2) = [\_instr \mapsto L, c \mapsto L\ chan]$</td><td>$L$</td><td>3</td></tr>
<tr><td>$G(3) = [\_instr \mapsto L, c \mapsto H\ chan]$</td><td>$L$</td><td>4</td></tr>
<tr><td>$G(1) = [\_instr \mapsto L, c \mapsto U\ chan]$</td><td>$L$</td><td>4</td></tr>
<tr><td>$G(4) = [\_instr \mapsto U, c \mapsto U\ chan]$</td><td>$L$</td><td>5</td></tr>
<tr><td colspan="4">Output : push(sup(top(pc), evalT(TypeOf_Expression($lowValue$))), pc);
if $lowValue$ then
$c := publicChannel$;
updateEnv(g_M, c, $G(2)(c)$);
else
$c := privateChannel$;
updateEnv(g_M, c, $G(3)(c)$);
end;
pop(pc);
tau = TypeOf_Expression(c);
$tau1$ = TypeOf_Expression($highValue$);
if$(((tau = L\,chan)\ and\ (\mathsf{sup}(\mathsf{evalT}(tau1), \mathsf{top}(pc)) = H))\ or\ (tau = B\ chan))$
then fail;
else send $highValue$ to c;</td></tr>
</table>

Fig. 3. The send of a high value on an unknown channel calls for instrumentation

The example presented in Figure 4 shows how the instrumentation algorithm works. The inference algorithm determines that the program needs instrumentation. The program is shown on the upper left corner of the figure. The instrumentation result is shown in the lower part of the figure. The third instruction receives a channel name on another one. The instrumentation is necessary to obtain the real type of this channel. In the sixth instruction of the instrumented program, the update of $G(3)$ is due to the fact that the inference algorithm marks the channel c as unknown on that line. The fourth instruction is a **send** command. A check is inserted in the instrumented code to ensure that a secret information is neither sent on a public channel (the type of c being unknown statically) nor on a blocked one (B).

A more "realistic" example is described in [22]: one may want to "prohibit a personal finance program from transmitting credit card information over the Internet even though the program needs Internet access to download stock market reports. To prevent the finance program from illicitly transmitting the private information (perhaps cleverly encoded), the compiler checks that the information flows in the program are admissible." This could be translated into the code of Figure 5 where all the channels, except *internet*, are private.

6 Related work

Securing flow information has been the focus of active research since the seventies. Dynamic techniques were the first methods as in [8]. Some of those tech-

Input to analyzer	Inference analysis		
	Environment	pc	i
receive$_{\mathbf{c}}$ v **from** $privateChannel$;	$G(1) = [\_instr \mapsto L, v \mapsto H\ val]$	L	2
if $lowValue$ **then**		$pc_{if} = L$	3
receive$_{\mathbf{n}}$ c **from** $publicChannel$;	$G(3) = [\_instr \mapsto L, v \mapsto H\ val, c \mapsto U\ chan]$	L	4
send v **to** c	$G(4) = [\_instr \mapsto U, v \mapsto H\ val, c \mapsto U\ chan]$	L	5
else skip	$G(5) = [\_instr \mapsto L, v \mapsto H\ val]$	L	6
end	$G(2) = [\_instr \mapsto U, v \mapsto H\ val, c \mapsto U\ chan]$	L	6

Output :
receive$_{\mathbf{c}}$ v **from** $privateChannel$;
updateEnv($g\_M, v, G(1)(v)$);
push(sup(top(pc), evalT(TypeOf_Expression($lowValue$))), pc);
if $lowValue$ **then**
  receive$_{\mathbf{n}}$ c **from** $publicChannel$;
  updateEnv($G(3), c$, TypeOf_Channel(c));
  updateEnv($g\_M, c, G(3)(c)$);
  tau = TypeOf_Expression(c);
  $tau1$ = TypeOf_Expression(v);
  if(($tau = L\ chan$) *and* (sup(evalT($tau1$), top(pc)) = H))
    or ($tau = B\ chan$))
    then fail;
    else send v **to** c;
else skip;
end;
pop(pc);

Fig. 4. The **send** of a high value on an unknown channel calls for instrumentation

Input	**receive**$_{\mathbf{c}}$ $stockMarketReports$ **from** $internet$; **send** $stockMarketReports$ **to** $screen$; **receive**$_{\mathbf{c}}$ $creditCardNumber$ **from** $settings$; **send** $creditCardNumber$ **to** $secureLinkToBank$; **receive**$_{\mathbf{c}}$ $latestTransactions$ **from** $secureLinkToBank$; **send** $latestTransactions$ **to** $screen$; $cleverlyEncodedCreditCardNumber := creditCardNumber * 3 + 2121311611218191$; **send** $cleverlyEncodedCreditCardNumber$ **to** $internet$
Output	Error (Send) : Cannot send $cleverlyEncodedCreditCardNumber$ (H) to $internet$ (L).

Fig. 5. Example from [22]

niques try to prevent explicit flows as well as implicit flows in program runs. Those techniques are given with no soundness proof. Denning and Denning [5] introduce for the first time, secure information-flow by static analysis, based on control and data flow analysis. Subsequently, many approaches have been devised using type systems. They vary in the type of language, its expressivness and the property being enforced. Volpano and Smith in [21] introduce a type based analysis for an imperative language. Pottier and Simonet in [16] analyse the functional language ML, with references, exceptions and polymorphism. Banerjee and Naumann devise a type based analysis for a Java-like language. Their analysis however has some trade-offs like low security guards for conditionals that involve recursive calls. In [14], Myers statically enforces information policies in JFlow, an extension of Java that adds security levels annotations to variables. Barthe et al. in [2], investigate logical-formulation of non-interference, enabling the use of theorem proving or model-checking techniques. Nevertheless, purely static approaches are too conservative and suffer from a large number of false positive. In fact some information need to take an accurate decision are often only available during execution. This has cause a revival of interest

for dynamic analysis. Russo and Sabelfeld in [18], prove that dynamic analyses could enforce the same security policy enforced by most static analyses, termination-insensitive non-interference and even be more permissive (with less false-positive). This is true for flow insensitive analyses but not for flow sensitive ones. In [17], Russo and Sabelfeld show the impossibility for a sound purely dynamic monitor to accept the same set of programs accepted by the classic flow sensitive analysis [11] of Hunt and Sands. Russo and Sabelfeld in [17] present a monitor that uses static analysis during execution. In [9], the authors present an interesting approach to non-interference based on abstract interpretation.

Our approach is flow sensitive, similarly to [11]. However, it distinguishes between variables in live memory and channels. We argue that our approach lead to less false positive and to lighter executions than existing approaches.

7 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, then it is secure according to the 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 their stored values. We clearly distinguish between variables and channels through which the program communicates, which is more realistic.

Our main contribution is the handling of a multi-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 extensions to take into account concurrency, declassification and information leakage due to termination [20, 1, 13]. We would like to scale up the approach to deal with real world languages and to test it on elaborate programs. The use of abstract interpretation to prove the correctness is also to be considered in a future work.

References

1. A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, Piscataway, NJ, USA, June 2012. IEEE Press.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2004.
3. A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi. Hybrid flow analysis implementation. `http://lsfm.ift.ulaval.ca/Recherche/Hybrid_analysis`, 2013.

4. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, May 1976.
5. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, July 1977.
6. J. Desharnais, E. P. Kanyabwero, and N. Tawbi. Enforcing information flow policies by a three-valued analysis. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security*, MMM-ACNS'12, pages 114–129, Berlin, Heidelberg, 2012. Springer-Verlag.
7. J. Desharnais, E. P. Kanyabwero, and N. Tawbi. Enforcing information flow policies by a three-valued analysis, long version. `http://www.ift.ulaval.ca/departement/professeurs/tawbi_nadia/`, 2012.
8. J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.
9. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. *SIGPLAN Not.*, 39(1):186–197, January 2004.
10. S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.
11. S. Hunt and D. Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, January 2006.
12. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
13. S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 881–893, New York, NY, USA, October 2012. ACM Press.
14. A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.
15. K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, July 2006.
16. F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the The ACM Symposium on Principles of Programming Languages*, 2002.
17. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.
18. A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research, 2009.
19. G. Smith. Principles of secure information flow analysis. In *Malware Detection*, volume 27, pages 291–307. Springer, 2007.
20. J. Thomas, N. Cuppens-Boulahia, and F. Cuppens. Declassification policy management in dynamic information systems. In *ARES 2011: 6th International Conference on Availability, Reliability and Security*, 2011.
21. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.
22. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15:2002, 2002.