Instruction-level security analysis for information flow in stack-based assembly languages

Nicoletta De Francesco, Luca Martini

Dipartimento di Ingegneria dell’Informazione: Elettronica, Informatica, Telecomunicazioni, Università di Pisa,
Via Diotisalvi, 2, 56126 Pisa, Italy

Received 6 September 2005; revised 21 December 2006
Available online 8 May 2007

Abstract

We propose a method to analyze secure information flow in stack-based assembly languages, communicating with the external environment by means of input and output channels. The method computes for each instruction a security level for each memory variable and stack element. Instruction-level security analysis is flow-sensitive and hence is more precise than other analyses, such as standard security typing. Instruction-level security analysis is specified in the framework of abstract interpretation. We define concrete operational semantics which handles, in addition to execution aspects, the flow of information of the program. The basis of the approach is that each value is annotated by a security level and that the abstract domain is obtained from the concrete one by keeping the security levels and forgetting the actual values. Operand stack are abstracted as fixed-length stacks of security levels. An abstract state is a map from instructions to abstract machine configurations, where values are substituted by security levels. The abstract semantics consists of a set of abstract rules manipulating abstract states. The instruction-level security typing can be performed by an efficient fixpoint iteration algorithm, similar to that used by bytecode verification.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Abstract interpretation; Flow-sensitive; Language-based security; Static analysis

1. Introduction

The secure information flow property within programs in multilevel secure systems requires that information at a given security level does not flow to lower levels [42]. The goal is to guarantee confidentiality: an observer that has access to information with a given secrecy level must be prevented from finding out anything about more secret information.

* This work was partially supported by the Italian COFIN 2004 project “AIDA: Abstract Interpretation Design and Application”.
* Corresponding author.
E-mail addresses: nico@iet.unipi.it (N. De Francesco), luca.martini@iet.unipi.it (L. Martini).
A common way to ensure confidentiality is based on access control mechanisms [12]: each user is given read and/or write right on a set of objects. However, access control mechanisms only control the release of information, they do not check the propagation of the information within the accessed entity. On the other hand, once a user has been given an access right to some data, by analyzing the information flow within programs it is possible to check whether the accessed information is being used properly, i.e. that it is not propagated to other entities who are not allowed to know that particular data. Let’s look at the following example in the context of Java Card.1

Example. A Java Card [18] is a smart card running a Java Virtual Machine (the Java Card Virtual Machine—JCVM), and is becoming a secure token in various fields, such as banking and public administration. The Java Card system was designed to speed up the development of applications (applets) and to increase portability. The JCVM is single-threaded, but more than one applet can coexist on the same card. Applets are normally isolated by means of the Java Card Firewall mechanism. This firewall allows an applet to access external objects only through an object sharing mechanism, called a shareable interface. The firewall is based on an access control policy and therefore does not control information propagation.

Now let’s look at a card that hosts three applets, say A, B, and C, each issued by a different commercial entity (see Fig. 1). Suppose that some partnership exists such that A must share some data with B to cooperate and likewise B with C in order to be able to cooperate, whereas A and C must not share information. Therefore the applets must be programmed to inform the firewall of this cooperation. Thus the firewall allows communication between A and B and communication between B and C, but prevents communication between A and C. Nevertheless, it is not able to check whether A’s confidential data is not being propagated to C through B. This can be checked by analyzing the information flow in B’s code.

This work deals with a static analysis of programs to check information flow. We consider stack-based assembly language programs communicating with the external environment by means of input and output channels, used during the execution to receive and to send data, respectively. Channels are the only way in which the program communicates with the external environment, while the internal behavior of the machine executing the program is not observable. A security policy associates each channel with a security (secrecy) level: a channel with a given secrecy level is allowed to contain only data with that level or a lower one. Security levels form a lattice. The program can be used by a set of users who are each given a fixed security level. A user is allowed to send/receive data to/from the program by using only channels with a security level less than or equal to the user level. A program satisfies the secure information flow property under a given security policy \( \mathcal{P} \) if the observation of the values sent to (received from) the channels having a given security level does not reveal any information about the values sent to (received from) the channels with higher (more secret) security levels. In other words, for each security level \( \sigma \), it must be the case that two executions of the program, receiving the same sequence of elements from the channels associated with \( \sigma \) or with a level lower than \( \sigma \), perform the same sequence of insertions and extractions to/from the channels associated with \( \sigma \) or with a level lower than \( \sigma \).

1 This example is inspired by the PACAP case study [17].
We present a static analysis to analyze information flow in a specific language. We call this analysis instruction-level security analysis. It is based on an abstract interpretation approach [20–22]. The program is abstractly executed on a domain of security levels instead of normal data. The abstract semantics is obtained as an abstraction of a concrete one. We define an instrumentation of the standard dynamic semantics of the language to take into account the security level of the data. The instrumented concrete operational semantics handles, in addition to execution aspects, the level of the flow of information of the program. Each value is annotated by a security level and the level of data flowing through the variables and the stack is calculated dynamically. We then define a collecting semantics [33] associating to each program point (instruction) the set of concrete states in which the machine can be when the point is reached. We prove that the collecting semantics correctly annotates data with the security level on which they depend.

We then define the abstract semantics and prove that it is a correct abstraction of the collecting semantics. The abstract execution always terminates and the output of the analysis is a mapping from instructions to abstract machine configurations. Each configuration specifies a security level for each memory location, stack element and channel. This level is the maximum security level of the information that can be present in the memory locations, stack and channel during any execution of the program when the corresponding instruction is executed. The program has secure information flow if the level computed by the analysis for each channel in each configuration is less than or equal to the level assigned to the channel by the security policy. The abstract execution of the program can be performed by means of an efficient fixpoint iteration algorithm, which was inspired by the Java Bytecode Verification [39]. Java Bytecode is the intermediate code produced by Java compilers. The Bytecode Verifier performs safety checks on the code by using an instruction-level typing algorithm. For each instruction it infers a type for every variable and stack element (assuming that the stack height is fixed for each instruction). Java Bytecode Verification is performed during the execution of a Java program whenever a Java class is required for execution. It must thus be efficient in order to not slow down the execution. Our analysis is similar to Java Bytecode Verification but it operates on security levels instead of types.

Let us summarize the main characteristics and contributions of the paper.

- We present an approach to check the secure information flow property, which is based on abstract interpretation. The abstract interpretation framework allows a standard formal proof of the correctness of the analysis. Moreover, since the program is executed, even though abstractly, it is generally possible to obtain a greater accuracy than with other static methods. The abstract interpretation which underlies our method is able to infer for each instruction the maximum security level of the data that can pass through the channels during any execution of the program. We obtain a greater precision than other methods that check secure information flow, for example those based on security typing.
- The method is a good compromise between expressive power and efficiency. It can be implemented by an algorithm similar to the well known very efficient one executed by the Java Bytecode Verifier.
- The secure information flow property that we define concerns the sequence of manipulations of the input and output channels during the execution. A program can also be seen as a process, possibly looping forever, which executes in parallel with other processes and communicates with them by means of the input and output channels. Hence, the method can be easily extended to concurrent languages.
- The language we consider is an assembly language. Additional problems with respect to high-level languages are required when checking assembly languages, above all with regard to stacks. We introduce a parametric notion of stack abstraction which can be used for assembly languages and allows more flexibility than existing ones.

We applied the method described in the paper to a high-level language with dynamic structures [23]. The initial idea was originally conceived by Avvenuti et al. [5].

The language and the security model are presented in Section 2, while Section 3 gives an overview of the method. Section 4 describes the concrete semantics and proves that is suitable for describing information flows. Section 5 introduces the abstract domains and their relation with the concrete ones. Section 6 presents the abstract semantics, that is the instruction-level security analysis, proves its correctness and discusses its complexity. An application of the method to a non-trivial example is presented in Section 7. Finally, Section 8 discusses related work and concludes.
2. The Model

We consider a representative subset of instructions of a conventional stack-based assembly language that manipulates integers. Our language has an operand stack, a memory containing the local variables, simple arithmetic instructions, conditional and unconditional jumps and primitives for input/output in a message passing style. The instructions are reported in Fig. 2, where \( x \) ranges over a set \( \text{Var} \) of local variables, \( \text{op} \) over a set of binary arithmetic and logic operations (\( \text{add, sub, eq...} \)) and \( a \) over a set of I/O channels.

Each instruction is labeled by a label \( t \in B = \{1, 2, \ldots, n\} \cup \{\text{end}\} \). The fake instruction \text{end} represents the termination of the program. Hereafter, \( \text{Names}_I \cap \text{Names}_O = \emptyset \). The set \( \text{Names} = \text{Names}_I \cup \text{Names}_O \) denotes the set of all channels.

We assume that programs respect the following constraints:

1. executions do not jump to undefined addresses;
2. no stack overflow and underflow occurs;
3. for each instruction, the height of the stack when the instruction is executed is always the same in all executions.

The first and second assumptions are made only for simplicity of presentation, in order to avoid performing the corresponding checks when defining the semantics of the language. The third one is not too restrictive: it is, for example, checked by the Bytecode Verifier [39, Section 4.9.2] for compiled Java programs. The first assumption can be statically verified by checking all the labels of if and goto instructions. A simple abstract interpretation based analysis can be used to statically check the second and the third assumptions: each instruction is abstracted as its effect on the stack height, and the analysis can calculate the stack height at each program point.

The run-time behavior, described informally in Fig. 2, is illustrated in Fig. 3 by means of a small-step operational semantics. A state \( \langle t, \mu^e, s^e, c^e \rangle \) of the computation is composed of:

- the label \( t \) of the instruction being executed;
- the memory \( \mu^e \) that maps variables names to integer values;
- the operand stack \( s^e \) which is a sequence of integer values;
- the channels status \( c^e \) that maps channel names to their contents.

Each channel is modeled as a sequence of integers. We use the \( \cdot \) operator to denote concatenation, \( \lambda \) for the empty sequence, and \( \sharp s \) for the length of the sequence (stack) \( s \). If the stack is \( s^e = [k_1 \cdot k_2 \cdot \ldots \cdot k_n] \), then \( k_1 \) is

\[
\begin{align*}
\text{op} & \quad \text{pop two operands off the stack, perform the operation and push the result onto the stack} \\
\text{pop} & \quad \text{discard the top value from the stack} \\
\text{push} k & \quad \text{push the constant } k \text{ onto the stack} \\
\text{load} x & \quad \text{push the value of the variable } x \text{ onto the stack} \\
\text{store} x & \quad \text{pop off the stack and store the value into variable } x \\
\text{send} a & \quad \text{pop off the stack and send the value to the output channel } a \\
\text{recv} a & \quad \text{receive a value from the input channel } a \text{ and push it onto the stack into variable } x \\
\text{if} j & \quad \text{pop off the stack and jump to } j \text{ if zero} \\
\text{goto} j & \quad \text{jump to } j \\
\text{halt} & \quad \text{stop}
\end{align*}
\]

Fig. 2. Instruction set.
Fig. 3. Dynamic semantics for the instruction set in Fig. 2.

The transitions defined by the rules are labeled with actions that can be of three types:

- input actions: $a \gg k$ with $a \in \text{Names}_I$ and $k \in \mathbb{Z}$, meaning that the value $k$ has been received from input channel $a$;
- output actions: $a \ll k$, with $a \in \text{Names}_O$ and $k \in \mathbb{Z}$, meaning that the value $k$ has been sent to output channel $a$;
- unobservable action: $\iota$, meaning that the channels are left unchanged.

The transitions have been labeled to emphasize that it is not possible from the outside to inspect the complete state of the computation, but only the observable actions, i.e. operations performed on the channels. An initial state of the program is given by means of the initial contents of the input channels. Given a configuration of input channels $c^I_f : \text{Names}_I \rightarrow \mathbb{Z}^*$, the initial state corresponding to $c^I_f$ is defined as $(1, \mu^0, \lambda, c^I_0)$, where $\mu^0$ associates the value 0 with every variable, the stack is empty, for every channel $a \in \text{Names}_I$: $c^I_0(a) = c^I_f(a)$ and $\forall b \in \text{Names}_O, c^I_0(b) = \lambda$, i.e. the initial state of the output channels is empty. Finally, we denote by $\gamma(c^I_f)$ the (finite or infinite) sequence of actions that label the transitions of the computation starting from the initial state corresponding to $c^I_f$.

The security model is based on a set of security (secrecy) levels. Security levels are defined as a finite lattice $(\mathcal{L}, \sqsubseteq)$, partially ordered by $\sqsubseteq$. The levels are ordered in increasing secrecy: higher levels represent more secret information. We assume that $\mathcal{L}$ contains a minimum element $\text{min}_\mathcal{L}$. Least upper bound operation (lub) is denoted by $\sqcup_\mathcal{L}$.

Given a program $P$, we denote a security policy by $P : \text{Names} \rightarrow \mathcal{L}$; a security policy is an assignment of security levels to the channels used in $P$. We assume that each external observer with a confidentiality level $\gamma \in \mathcal{L}$ can only inspect the channels with a secrecy level less than or equal to $\gamma$. The secure information flow property that we want to ensure is that the information belonging to a security level is not affected by the information belonging to higher security levels. More precisely, for any channel $a$, the contents of $a$ in any computation

<table>
<thead>
<tr>
<th>Execution Values</th>
<th>$c^e \in \mathbb{Z}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memories</td>
<td>$\mu^e \in M^e = \text{Var} \rightarrow \mathbb{Z}$</td>
</tr>
<tr>
<td>Stacks</td>
<td>$s^e \in S^e = \mathbb{Z}^*$</td>
</tr>
<tr>
<td>Channels</td>
<td>$c^e \in C^e = \text{Names} \rightarrow \mathbb{Z}^*$</td>
</tr>
</tbody>
</table>

Fig. 4. Domains of the dynamic semantics.
must not be affected by data from channels with a security level higher than \( \mathcal{P}(a) \). Thus external observers with secrecy level \( \mathcal{P}(a) \) cannot know the more secret information, because they can only inspect the channels with levels that are lower than or equal to \( \mathcal{P}(a) \). The property is formally defined as follows.

**Definition 2.1 (secure information flow).** Let \( P \) be a program and \( \mathcal{P} \) a security policy for \( P \).

Given \( \sigma \in \mathcal{L} \), let us denote by \( \text{Names}^\mathcal{F}_\sigma \setminus \mathcal{P} = \{ a \mid a \in \text{Names}_I, \mathcal{P}(a) \subseteq \sigma \} \) and \( \text{Names}^\mathcal{F}_\sigma = \{ a \mid a \in \text{Names}_O, \mathcal{P}(a) \subseteq \sigma \} \) the set of input, respectively, output channels associated by \( \mathcal{P} \) to a security level less than or equal to \( \sigma \).

Let \( c_{I,1}, c_{I,2} : \text{Names}_I \rightarrow \mathbb{Z}^\times \) be any two initial states of the input channels that agree on channels with security level less than or equal to \( \sigma \), i.e. \( \forall a \in \text{Names}_I^\mathcal{F} : c_{I,1}(a) = c_{I,2}(a) \).

Let the sequences \( \gamma_2^1 \) and \( \gamma_2^2 \) be the projections of \( \gamma(c_{I,1}) \) and \( \gamma(c_{I,2}) \), respectively, on the actions in \( \text{Names}_I^\mathcal{F} \cup \text{Names}_O^\mathcal{F} \), that is the actions concerning only the input and output channels with a security level lower than or equal to \( \sigma \).

\( P \) has \( \sigma \)-secure information flow \((\sigma\text{-secure})\) under \( \mathcal{P} \) if \( \gamma_2^1 = \gamma_2^2 \).

\( P \) has insecure information flow \((\sigma\text{-insecure})\) if it is \( \sigma \)-secure for each \( \sigma \in \mathcal{L} \).

For each security level \( \sigma \), \( \sigma \)-security assures that, for any two computations that agree on the initial contents of the input channels with security level less than or equal to \( \sigma \), the sequence of data sent to/received from the input/output channels with security level less than or equal to \( \sigma \) is the same. General security means \( \sigma \)-security for every \( \sigma \).

3. Overview of the method

Instruction level security analysis is based on abstract interpretation [20–22]. Abstract interpretation is a method for analyzing programs in order to collect approximated information about their run-time behavior. It is based on a non-standard abstract semantics, which is a semantic definition in which a simpler (abstract) domain replaces the standard (concrete) one, and the operations are interpreted on the new domain. The static analysis is the abstract semantics.

The abstract semantics operates on security levels instead of actual values: each value is abstracted as a security level. The security level of a value is the lub of the security levels of all information flows on which the value depends. The contents of each channel is abstracted as a security level, initially the level associated with the channel by the security policy. The levels flowing through the variables, the stack, and the channels are calculated dynamically, taking into account both the explicit and the implicit information flows. An explicit flow occurs when a memory location is assigned a value or when a value is taken from (inserted onto) a channel or when a value is pushed onto the stack. An implicit flow occurs when an instruction may or may not be executed depending on some condition. Implicit flows are introduced by conditional commands. For example, the high-level language command \( \text{if } (y=0) \text{ then } x:=0; \text{ else } \text{skip} \) introduces an implicit flow from \( y \) to \( x \). In fact, checking the final value of \( x \) reveals information on the value of \( y \). Hence, the level of \( x \) after the execution of the command must take into account the level of \( y \). To manage implicit flows, we use a security environment, which associates a security level with each instruction. Whenever a conditional jump instruction \( t \) is executed, the level of the condition (top of the stack) is used to upgrade the environment of all instructions belonging to the scope of \( t \). When an instruction is executed, the levels manipulated by that instruction are upgraded, when necessary, to the level of the environment.

In the abstract semantics, security levels are manipulated instead of values, according to the semantics of the instructions and to the environment. For example, a \( \text{recv } a \) instruction at address \( t \) pushes onto the stack the lub of the level contained in \( a \) and the level of the environment of \( t \). Also the contents of \( a \) can be upgraded according to the level of the environment of \( t \). A \( \text{send } a \) instruction (possibly) upgrades the contents of \( a \) taking into account both the level which is the top of the stack and the level of the environment. We use abstract stacks of parametric length. If \( n \) is the maximum length of abstract stacks, a concrete stack of length \( n \geq n \) is approximated by an abstract stack in this way: its first \( n-1 \) elements model the first \( n-1 \) items of the concrete stack and the last one approximates all the remaining \( m-n+1 \) concrete items.
An abstract state is a mapping $q$, associating to each instruction $t$ an abstract machine configuration $q(t)$ that contains security levels instead of actual values. The configuration $q(t)$ represents the abstract machine state before the execution of $t$. Each configuration describes also the environment level of the corresponding instruction. Instruction $t$ is executed in state $q(t)$ and the produced state is merged (by means of a lub operation) with the state $q(t')$ of each successor instruction $t'$, producing a new state for $q(t')$. Merging is necessary since, due to the conditional and unconditional jumps, there are instructions corresponding to a join of different paths of the control flow graph. Merging two states consists in merging the level of each variable, stack element and channel. The analysis iterates the execution of the program instructions until a fixpoint is reached, that is $q$ is no longer modified. The final state obtained is the output of the analysis. The secure information flow property is satisfied if the contents of each channel in each row are less than or equal to the level assigned to the channel by the security policy.

In order to apply the abstract interpretation approach, we need a concrete semantics to be abstracted onto the abstract one. The concrete semantics is obtained by instrumenting the standard dynamic semantics to take into account the security level of the data. The instrumented concrete operational semantics handles, in addition to execution aspects, the level of the flow of information of the program. In fact it operates on values annotated with security levels. We then define a collecting semantics [33] associating to each program point (instruction) the set of concrete states in which the machine can be when that point is reached. We prove that the collecting semantics correctly annotates data and thus is able to dynamically reveal violations of secure information flow. The concrete semantics is used only for proving the correctness of the analysis. If we do not consider security annotation, this semantics reduces to the standard dynamic semantics defined in the previous section. We show that the abstract semantics is an abstraction of the collecting semantics, where the abstract domains are obtained from the concrete ones by keeping the security annotations and forgetting the actual values. The correctness of the method is then proved in the standard way for abstract interpretation.

### 3.1. An example

Let us now give an example of application of the analysis. Suppose that the lattice $\mathcal{L}$ only consists of two levels $L$ and $H$, with $L \sqsubseteq H$, and the program manipulates two input channels $\text{pubI}$ and $\text{privI}$, public and private, respectively, that is $P(\text{pubI}) = L$, $P(\text{privI}) = H$, and a public output channel $\text{pubO}$, such that $P(\text{pubO}) = L$. Consider the code in Fig. 5 which shows a non-secure implicit flow. The value taken initially from the private channel $\text{privI}$ affects the public channels $\text{pubI}$ and $\text{pubO}$. If it is different from 0, a value is taken from $\text{pubI}$ and sent onto $\text{pubO}$. If it is equal to 0, the channel $\text{pubI}$ is not used and 0 is put onto $\text{pubO}$. A user with level $L$, who is only allowed to observe the low channels, can then derive information on the value taken from $\text{privI}$.

```
1 recv privI   x = recv(privI);
2 if          if (x) {
3 recv pubI   y = recv(pubI);
4 store y    }
5 goto 8      else {
6 push 0      y = 0;
7 store y     }
8 load y      send(y, pubO);
9 send pubO
10 halt
```

Fig. 5. A simple example. The source code on the right (b) can be compiled in the assembly code on the left (a) by allocating the variable $x$ on the stack.
(zero or non-zero) by observing if a value has been taken or not from pubI or by observing the value inserted into pubO.

The initial state of the analysis is shown in Fig. 6(a). It is a table $q$ with 10 rows, each one corresponding to an instruction. In the first row the contents of the channels is that defined by the policy for input channels ($H$ for privI, $L$ for pubI) and the minimum level $L$ for the output channel pubO and for variable $y$. If we consider abstract stacks with maximum length 2, the initial stack, which is empty, is represented by $[\bot_V \cdot \bot_V]$, where $\bot_V$ is the bottom abstract value. In the other rows of the table, $y$ and the stack hold undefined values (respectively, $\bot_V$ and $\bot_S$) and all the channels are set to the minimum level $L$. In each row the environment is set to the minimum level $L$. Fig. 6(b) shows the state after instruction 1 has been abstractly executed. The machine state produced by the execution of instruction 1 (an item taken from channel privI has been pushed onto the stack) is merged with $g(2)$ (the old machine state corresponding to instruction 2). Note that the stack contains $H$, since the datum taken from a channel coincides with the level of the channel. When instruction 2 is executed, since the top of the stack is $H$, the environment of all instruction in the scope of 2 (i.e. 3, 4, 5, 6, 7) is upgraded to $H$, to indicate the implicit flow generated by instruction 2. The machine configuration after the execution of instruction 2 is merged with rows 3 and 8, corresponding to the instructions which are immediate successors of 2; the resulting table is shown in Fig. 6(c). When instruction 3 is executed, channel pubI contains $L$ but, since instruction 3 has a high environment, the level put onto the stack is $H$ and the level of channel pubI is upgraded to $H$, see Fig. 6(d). After executing the subsequent instructions, the level of $y$ becomes $H$ and in the final state (the fixpoint, see Fig. 6(e)) the level of pubO is $H$ too. Here, the level computed for channels pubI and pubO is higher than that assigned to them by the security policy, and so the program is not certified by the analysis.

The analysis reveals what channels may contain information that is not allowed to flow through them. Hence we are able to identify the insecure points of the program and in particular the insecure channels. In fact we compute the maximum security level of the data taken from/sent to each channel, starting from the level assigned to input channels by the security policy.

4. Concrete semantics

In this section, we define the concrete semantics of the language. To take into account the security level of data, we annotate each value $k \in \mathbb{Z}$ with a security level, representing the lub of the security levels of the explicit and implicit information flows on which $k$ depends. Hence a concrete value is a pair $(k, \sigma)$, where $k$ is an integer and $\sigma$ a security level. A memory is a map from variable names to concrete values. A stack is a sequence of concrete values. Each channel is represented by a pair whose first element is a sequence of integer values (i.e. the sequence of values present on the channel), and the second is a security level that represents the current security level of the channel. This level, initialized by the policy, can be upgraded during the execution depending on the operations issued on the channel. The concrete domains are defined in Fig. 7.

The semantics is defined by means of the set of rules in Fig. 8, defining a labeled relation $\rightarrow : Q \rightarrow Q$ between the states of the computation. If security annotations are ignored the rules are the same as those of the standard dynamic semantics presented in Section 2. Moreover, a state is still composed by a memory, a stack, and channels, (that now manipulate concrete values) plus a security environment $\rho$. The set of states is then $Q = B \times Env \times M \times S \times C$. Each state $q \in Q$ is a tuple $\langle t, \rho, \mu, s, c \rangle$ that describes the configuration of the machine when executing the instruction $t$: $\mu$ defines the values of variables, $s$ represents the status of the stack, while $c$ describes the channels status. The security environment $\rho \in Env$ in each state associates a security level with every instruction. This security level represents the level of the implicit flow under which the instruction is executed. The environment is initially set to $min_E$ for all instructions and can be upgraded by the conditional jump instructions.

Values manipulated under a security environment are upgraded when necessary, as follows. Each value $(k, \tau)$ that has been evaluated, tested, pushed onto the stack, stored in a variable, sent or received during the execution of instruction $t$, changes its security level into $\rho(t) \sqcup C \tau$. In rule $\text{op}$ the level of the value pushed onto the stack is the lub of the levels of the used values and of the environment. In rule $\text{push}$ the level of the constant value pushed onto the stack is the level of the environment, since the initial level of a constant is assumed to be the minimum secrecy level. Rule $\text{load}$ and $\text{store}$ are similarly defined.
Rule **receive** takes a value from the specified input channel and pushes it onto the stack, annotated with the lub of the level of the channel and the environment of $t$. The secrecy annotation of the channel is updated in the
As a consequence, if $\rho(t)$ is higher than the previous annotation of the channel, then the annotation of the channel is upgraded, to indicate the fact that the manipulation of the channel depends on an information flow with level higher than the previous annotation. Likewise, in rule send, the level of the specified output channel can be upgraded taking into account the level of the value and that of the environment of the instruction.

Whatever branch is chosen, rules if upgrade the environment of all the instructions belonging to the scope of the conditional jump instruction, in order to take into account the level of the condition. Given a conditional instruction $t$, the set $\text{scope}(t)$ contains all the instructions that may or may not be executed depending on the value tested by $t$. The definition of $\text{scope}$ is given in the following subsection.

### 4.1. Definition of scope

We denote by $\succ \subseteq B \times B$ the relation where $t \succ t'$ says that instruction $t'$ can be executed immediately after instruction $t$, that is

- $t \succ j$ if $t: \text{goto } j$
- $t \succ j, t \succ t + 1$ if $t: \text{if } j$
- $t \succ \text{end}$ if $t: \text{halt}$
- $t \succ t + 1$ otherwise

Moreover, we define $\succ^*$ as the transitive closure of $\succ$. We also need a function $\text{ipd}: B \rightarrow B$ that calculates the immediate postdominator of an instruction. The immediate postdominator of $t$ represents the first common instruction in all the possible execution paths that start from $t$ and reach the final node [16].

**Definition 4.1 (Postdomination, immediate postdomination).** Let $t_1, t_2$ be instructions in $B$: $t_2$ postdominates $t_1$, denoted by $t_2 \text{ pd } t_1$, if $t_2$ is on every possible execution path from $t_1$ to $\text{end}$. We use $t_2 \text{ pd}^+ t_1$ iff $t_1 \text{ pd } t_2$ and $t_1 \neq t_2$. Moreover, $t_2$ immediately postdominates $t_1$, denoted by $t_2 = \text{ipd}(t_1)$, if $t_2 \text{ pd}^+ t_1$ and there is no instruction $t_3$ such that $t_2 \text{ pd}^+ t_3 \text{ pd}^+ t_1$.

For each conditional jump instruction $t$, the set $\text{deps}(t)$ contains all the instructions of the control flow graph between $t$ and $\text{ipd}(t)$:

$$\text{deps}(t) = \{t' | \exists \text{ a path from } t \text{ to } t' \text{ that does not contain } \text{ipd}(t)\}$$

Since we analyze non-terminating programs as well as terminating ones, we must take into account that, if a loop exists in $\text{deps}(t)$, then it is possible that the instructions following $\text{ipd}(t)$ are not executed if the program loops infinitely. Hence also these instructions are affected by the implicit flow generated by $t$. Moreover, it is also possible that the paths starting from $t$ never join, since each branch is a loop and $\text{ipd}(t)$ does not exists. Also in this case, all the instructions that are reachable from $t$ must be considered in the scope of $t$. The scope of the implicit flow generated by $t$ is defined by the function $\text{scope}: B \rightarrow \wp(B)$.

**Definition 4.2 (Scope).**

$$\text{scope}(t) = \begin{cases} \{t' | t \succ^* t'\} & \text{if } t \neq \text{end} \land \exists \text{ path } \pi : t \cdots t' \cdots t' \text{ s.t. } \text{ipd}(t) \notin \pi \\ \text{deps}(t) & \text{otherwise} \end{cases}$$
Fig. 8. Concrete semantics for the instruction set in Fig. 2.

If either $t$ does not reach end or $\text{deps}(t)$ contains a loop, then $\text{scope}(t)$ contains all the instructions reachable from $t$. On the other hand, if $\text{deps}(t)$ does not contain a loop, then $\text{scope}(t)$ is equal to $\text{deps}(t)$. The functions $\text{ipd}$, $\text{deps}$, and $\text{scope}$ can be statically computed using the control flow graph of the program [2].

### 4.2. Collecting semantics

We now define a collecting semantics [33], whereby each instruction is associated with the set of states in which the instruction can be executed in any computation. As a consequence, a concrete state is a collection of execution states. The concrete state domain is defined as follows. First, we formally define the initial state of a computation.

**Definition 4.3 (initial state).** Let $P$ be a program and $\mathcal{P}$ a security policy for $P$. Given an initial configuration $\text{init}_I : \text{Names}_I \rightarrow \mathbb{Z}^\mathcal{X}$ defining the contents of the input channels, the initial state is defined as $q(\text{init}_I) = (1, \rho_0, \mu_0, \lambda, c_0)$, where $\forall l \in B : \rho_0(l) = \min\mathcal{L}$, $\forall x \in \text{Var} : \mu_0(x) = (0, \min\mathcal{L})$, the stack is empty, $\forall a \in \text{Names}_I : c_0(a) = (c_I(a), \mathcal{P}(a))$, $\forall b \in \text{Names}_O : c_0(b) = (\lambda, \min\mathcal{L})$. 

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Rule</th>
<th>State Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{op}$</td>
<td>$t : \text{op}$</td>
<td>$\langle t, \rho, \mu, (k_1, \sigma_1) \cdot (k_2, \sigma_2) \cdot s, c \rangle \xrightarrow{\text{op}} \langle t + 1, \rho, \mu, (k_1 \text{ op } k_2, \sigma_1 \cup \mathcal{L} \cup \rho(t)) \cdot s, c \rangle$</td>
</tr>
<tr>
<td>$\text{push}$</td>
<td>$t : \text{push } k$</td>
<td>$\langle t, \rho, \mu, s, c \rangle \xrightarrow{\text{push } k} \langle t + 1, \rho, \mu, (k, \rho(t)) \cdot s, c \rangle$</td>
</tr>
<tr>
<td>$\text{pop}$</td>
<td>$t : \text{pop}$</td>
<td>$\langle t, \rho, \mu, \nu \cdot s, c \rangle \xrightarrow{\text{pop}} \langle t + 1, \rho, \mu, s, c \rangle$</td>
</tr>
<tr>
<td>$\text{load}$</td>
<td>$t : \text{load } x, \mu(x) = (k, \sigma)$</td>
<td>$\langle t, \rho, \mu, s, c \rangle \xrightarrow{\text{load } x, \mu(x)} \langle t + 1, \rho, \mu, (k, \sigma \cup \mathcal{L} \rho(t)) \cdot s, c \rangle$</td>
</tr>
<tr>
<td>$\text{store}$</td>
<td>$t : \text{store } x$</td>
<td>$\langle t, \rho, \mu, (k, \sigma) \cdot s, c \rangle \xrightarrow{\text{store } x} \langle t + 1, \rho, \mu, {x \leftarrow (k, \sigma \cup \mathcal{L} \rho(t))}, s, c \rangle$</td>
</tr>
<tr>
<td>$\text{send}$</td>
<td>$t : \text{send } a, c(a) = (s_1, \tau)$</td>
<td>$\langle t, \rho, \mu, (k, \sigma) \cdot s, c \rangle \xrightarrow{\text{send } a, c(a)} \langle t + 1, \rho, \mu, { [a \leftarrow (k, \sigma \cup \mathcal{L} \rho(t)) }, s, \tau \rangle \rangle$</td>
</tr>
<tr>
<td>$\text{receive}$</td>
<td>$t : \text{recv } b_k, c(b) = (k \cdot s_1, \sigma)$</td>
<td>$\langle t, \rho, \mu, s, c \rangle \xrightarrow{b_k \cdot k} \langle t + 1, \rho, \mu, (k, \sigma \cup \mathcal{L} \rho(t)) \cdot s, \tau \rangle [b \leftarrow (s_1, \sigma \cup \mathcal{L} \rho(t))]$</td>
</tr>
<tr>
<td>$\text{goto}$</td>
<td>$t : \text{goto } j$</td>
<td>$\langle t, \rho, \mu, s, c \rangle \xrightarrow{\text{goto } j} \langle j, \rho, \mu, s, c \rangle$</td>
</tr>
<tr>
<td>$\text{if}$</td>
<td>$t : \text{if } j, k \neq 0$</td>
<td>$\langle t, \rho, \mu, (k, \sigma) \cdot s, c \rangle \xrightarrow{\text{if } j, k \neq 0} \langle t + 1, \rho, \mu, [t' \in \text{scope}(t) \leftarrow (\rho(t') \cup \mathcal{L} \sigma)], \mu, s, c \rangle$</td>
</tr>
<tr>
<td>$\text{if}$</td>
<td>$t : \text{if } j, k = 0$</td>
<td>$\langle t, \rho, \mu, (k, \sigma) \cdot s, c \rangle \xrightarrow{\text{if } j, k = 0} \langle j, \rho, t' \in \text{scope}(t) \leftarrow (\rho(t') \cup \mathcal{L} \sigma)], \mu, s, c \rangle$</td>
</tr>
<tr>
<td>$\text{halt}$</td>
<td>$t : \text{halt}$</td>
<td>$\langle t, \rho, \mu, s, c \rangle \xrightarrow{\text{halt}} \langle \tau, \rho, \mu, s, c \rangle$</td>
</tr>
</tbody>
</table>
Definition 4.4 (concrete state domain of the abstract interpretation). The concrete state domain of the abstract interpretation is the complete lattice $(\wp(Q), \subseteq)$ where $\wp$ is the powerset operator and $\subseteq$ the ordering relation. This lattice has $\emptyset$ as the bottom element, $Q$ as the top element, the least upper bound and greatest lower bound operators are $\cup$ and $\cap$, respectively.

Given $Q \subseteq Q$, $\text{lift}(\{t, \rho, \mu, [v_0 \cdots v_n], c \}, Q) = \{t, \rho', \mu', [v'_0 \cdots v'_n], c'\}$ such that:

- $\forall t' \in B : \rho'(t') = \max_E(Q(t), t')$
- $\forall x \in \text{Var} : \mu'(x) = \text{up}(\mu(x), \max_M(Q(t), x))$
- $i = 0 \ldots n, v'_i = \text{up}(v_i, \max_S(Q(t), i))$
- $\forall a \in \text{Names} : c'(a) = \text{up}(c(a), \max_C(Q, a))$

$\text{lift}(Q) = \left( \bigcup_{q \in Q} \text{lift}(q, Q) \right) \cup Q$

Fig. 9. The lift and the max functions.

Definition 4.5 (concrete next operator). Given a set $Q \subseteq Q$ of concrete states, the application of the next operator yields the set of lifted states that are either in $Q$, or can be reached in one step of computation, starting from the states in $Q$:

$$\text{next}(Q) = \text{lift} \left( Q \cup \left\{ q \mid \exists q' \in Q : q' \xrightarrow{t} q \right\} \right)$$

Proposition 4.6 (monotonicity of next). next is monotonic in $(\wp(Q), \subseteq)$.

Proof. Straightforward by definition of next. □

Definition 4.7 (collecting semantics). The concrete collecting semantics $\text{sem} \in \wp(Q)$ is the least upper bound in $(\wp(Q), \subseteq)$ of the following increasing chain, defined for all $n \in \mathbb{N}$:

$$\text{sem}_0 = \left\{ q(i_{i_0}) \mid \forall i_0 \in \text{Names}_f \to \mathbb{Z}^* \right\}$$
$$\text{sem}_{n+1} = \text{next}(\text{sem}_n)$$

The function lift at each step aligns the security annotations to the top in order to properly manage implicit flows. These are the security annotations of the states corresponding to the join point of different branches of...
a conditional instruction. This function is necessary for the soundness of the concrete semantics, proved by the Theorem 4.8.

Example. Let us show that, if lift was not used by next, then the collecting semantics would not represent all the information flows. Consider the program in Fig. 10(a), whose corresponding high-level code is shown in Fig. 10(b). The program does not respect the secure information flow property, since the value inserted into the public channel pubO depends on the value read from the private input channel privI: this value is 1 if the contents of privI is 0, 3 otherwise.

There are two possible executions of the program, one when the value taken from privI is 0 and the other when it is $\neq 0$. In the first case the false branch of the first conditional jump instruction (labeled by 8) and the true branch of the second conditional jump instruction (labeled 12) are executed. In the second case, the computation follows the true branch of 8 and the false branch of 12. The collecting semantics collects two kinds of states at instruction 11 = ipd(8):

1. the state where the memory is: $[\mu(x) = (1,L), \mu(y) = (0,H), \mu(z) = (1,L)]: x$ contains a low value since the assignment to $x$ has not been executed;
2. states where the memory is: $[\mu(x) = (0,H), \mu(y) = (k \neq 0,H), \mu(z) = (1,L)]: x$ contains a high value since the assignment to $x$ has been executed under a high environment.

Let us now consider the second if at instruction 12: if it is executed starting from the state in (1), the condition is true and the environment of the instructions in scope(12) is $L$: hence $z$ is assigned $(3,L)$ (instruction 14). If instruction 12 is executed starting from the states in (2), since the condition is false, the assignment to $z$ is not executed and hence $z$ maintains the value $(1,L)$. Hence, in the memories belonging to all the states collected at instruction 15, $z$ is annotated with a low value. The send operation at instruction 16 is performed in any case in low environment, since 16 $\notin$ scope(12). As a consequence, the level of channel pubO, on which the value of

```
1 recv privI y=recv(privI);
2 store y x=1;
3 push 1 z=1;
4 store z if (y) {
  5 push 1 x=0;
  6 store x }
7 load y if (x) {
  8 if 11 z=3;
  9 push 0 }
10 store x send(z,pubO);
11 load x
12 if 15
13 push 3
14 store z
15 load z
16 send pubO
17 halt
```

Fig. 10. Example that shows the need for the function lift.
z is sent, is not upgraded and remains L in any case. Thus the collecting semantics does not reveal any incorrect flow on channel pubO. The lift operation adds to the states collected at instruction 11 the state equal to that in (l) except for the level of x, that is \([\mu(x) = (1, H), \mu(y) = (0, H), \mu(z) = (1, L)]\). This state represents the fact that, even though x has not been assigned during the execution of the if, its value, when the scope of the if is exited, depends on the high condition of the if. Hence, instruction 16 is also executed starting from this state, and channel pubO is upgraded to H. Note that, if all four combinations of the two branches of the two conditional jump instructions were possible, the lift operation would not be necessary. In fact it is not used by the abstract semantics, as it will be shown in Section 6, since the abstract semantics follows all the paths of the control flow graph, irrespectively of the fact that some computation may never occur.

The following theorem states that the annotation of data performed by the concrete semantics gives a sufficient criterion to ensure the information flow property, since it correctly associates values with the secrecy level on which they depend (see Section 4.3 for the proof).

**Theorem 4.8** (Soundness of the concrete semantics). *A program P has a secure information flow under a security policy P if for each concrete state \(\langle t, \rho, \mu, s, c \rangle \in \text{sem} \) for each channel a \(\in \text{Names} \), if \(c(a) = (\delta, \sigma)\), then \(\sigma \subseteq P(a)\).*

This theorem gives only a sufficient condition for secure information flow, i.e. not all secure programs satisfy the hypotheses of the theorem. Consider, for instance, the compiled version of this high-level code

```plaintext
y = recv(privI);
x = 1;
z = 1;
if (y) {
    send(x, pubO);
} else {
    send(z, pubO);
}
```

where privI is a private input channel and pubO is a public output channel. Since pubO is manipulated in a high environment, Theorem 4.8 cannot be applied to this program, even though it clearly has secure information flow since the same value is sent to the output channel in all possible executions. However, we believe that when standard compiler optimizations (like constant propagation and redundancy elimination) are applied these cases are very uncommon to occur.

### 4.3. Proof of Theorem 4.8

We start by defining a relation of \(\sigma\)-equivalence between concrete domains.

**Definition 4.9** (\(\sigma\)-equivalence). Let \(\sigma \in L\).

- Two concrete values \(v_1 = (k_1, \sigma_1)\) and \(v_2 = (k_2, \sigma_2)\) are \(\sigma\)-equivalent \(\left( v_1 \stackrel{\gamma}{=}^\sigma v_2 \right)\) if one of the following cases holds:
  - \(\sigma_1 \subseteq L \sigma\) and \(\sigma_2 \subseteq L \sigma\) and \(k_1 = k_2\)
  - \(\sigma_1 \not\subseteq L \sigma\) and \(\sigma_2 \not\subseteq L \sigma\)

- Two environments \(\rho_1\) and \(\rho_2\) are \(\sigma\)-equivalent \(\left( \rho_1 \stackrel{\gamma}{=}^\sigma \rho_2 \right)\) iff

\[\forall t \in B: \text{ either } \rho_1(t) \subseteq L \sigma\text{ and } \rho_2(t) \subseteq L \sigma\text{ or } \rho_1(t) \not\subseteq L \sigma\text{ and } \rho_2 \not\subseteq L \sigma\).

- Two concrete memories \(\mu_1\) and \(\mu_2\) are \(\sigma\)-equivalent \(\left( \mu_1 \stackrel{\gamma}{=}^\sigma \mu_2 \right)\) iff

\[\forall x \in \text{Var}: \mu_1(x) =^\gamma \mu_2(x)\).

- Two concrete stacks \(s_1\) and \(s_2\) are \(\sigma\)-equivalent \(\left( s_1 \stackrel{\gamma}{=}^\sigma s_2 \right)\) iff \(s_1[i] =^\gamma s_2[i]\) and \(\forall i \in \{1..\sharp s_1\} : s_1[i] =^\gamma s_2[i]\).

- Two concrete channels \(a_1 = (\delta_1, \sigma_1)\) and \(a_2 = (\delta_2, \sigma_2)\) are \(\sigma\)-equivalent \(\left( a_1 \stackrel{\gamma}{=}^\sigma a_2 \right)\) if one of the following cases hold:

\[\sigma_1 \subseteq L \sigma, \sigma_2 \subseteq L \sigma\text{ and } \delta_1 = \delta_2\]

\[\sigma_1 \not\subseteq L \sigma\text{ and } \sigma_2 \not\subseteq L \sigma\]

- Two concrete states \(q_1 = (t_1, \rho_1, \mu_1, s_1, c_1)\) and \(q_2 = (t_2, \rho_2, \mu_2, s_2, c_2)\) are \(\sigma\)-equivalent \(\left( q_1 \stackrel{\gamma}{=}^\sigma q_2 \right)\) iff

\[\forall t \in B: \text{ either } \rho_1(t) \subseteq L \sigma\text{ and } \rho_2(t) \subseteq L \sigma\text{ or } \rho_1(t) \not\subseteq L \sigma\text{ and } \rho_2 \not\subseteq L \sigma\).

\[\forall x \in \text{Var}: \mu_1(x) =^\gamma \mu_2(x)\).

\[\forall i \in \{1..\sharp s_1\} : s_1[i] =^\gamma s_2[i]\).

\[\forall \ell_1, \ell_2, \delta_1, \delta_2, \sigma_1, \sigma_2: \ell_1 \delta_1 =^\gamma \ell_2 \delta_2 \text{ where } \ell_1, \ell_2 \text{ are channels, } \delta_1, \delta_2 \text{ are actions, and } \sigma_1, \sigma_2 \text{ are secrecy levels}\).
The following lemma states that (a) the execution of a non-conditional jump instruction preserves \(\sigma\)-equivalence; and (b) \(\sigma\)-equivalence is also preserved with a conditional jump instruction, provided that the tested condition is less secret than \(\sigma\).

**Lemma 4.10.** Let be \(q_1 = (t, \rho_1, \mu_1, s_1, c_1) = \alpha q_2 = (t, \rho_2, \mu_2, s_2, c_2)\). If \(q_1 \xrightarrow{\ell} q'_1 = (t'_1, \rho'_1, \mu'_1, s'_1, c'_1)\) and \(q_2 \xrightarrow{\ell} q'_2 = (t'_2, \rho'_2, \mu'_2, s'_2, c'_2)\), then

\[
\rho'_1 =_\sigma \rho'_2, \quad \mu'_1 =_\sigma \mu'_2, \quad s'_1 =_\sigma s'_2, \quad \forall a \in \text{Names} : c'_1(a) =_\sigma c'_2(a)
\]

Moreover, if

- \(t\) is not a conditional jump instruction; or
- \(t\) is a conditional jump instruction and the top element of \(s_1\) has a security annotation \(\subseteq \mathcal{L} \sigma\).

**Proof.** By examining all possible kinds of instructions.

We can now prove the main theorem.

**(Proof of Theorem 4.8.)** Consider two concrete executions starting from the same configurations of the input channels belonging to \(\text{Names}^E\). Until a conditional jump instruction has not been reached with a high guard (i.e. top of the stack \(\mathcal{L} \mathcal{L} \sigma\)), by Lemma 4.10 the two executions perform the same instructions and reach at each step \(\sigma\)-equivalent states. In fact, if an input or output instruction is executed on a channel \(a\) such that \(\mathcal{P}(a) \subseteq \mathcal{L} \sigma\), by the hypothesis the corresponding value is the same. If no conditional jump with high guard is reached, the property is satisfied. If instead a conditional command is reached, say \(t\), with a high guard, then the two executions can be made up of different sequences of instructions, which may lead to not \(\sigma\)-equivalent states. Let \(q_1 = (t, \rho_1, \mu_1, t_1, s_1, c_1)\) and \(q_2 = (t, \rho_2, \mu_2, t_2, s_2, c_2)\) the states reached by the two computations, respectively, before the execution of \(t\) and let \(q'_1 = (t'_1, \rho'_1, \mu'_1, t_1, s_1, c_1)\) and \(q'_2 = (t'_2, \rho'_2, \mu'_2, t_2, s_2, c_2)\) the states after the execution of the instruction \(t\). By the \(\text{If rules of the semantics it holds that} \rho'_1 =_\sigma \rho'_2, \mu'_1 =_\sigma \mu'_2, \text{and for each instruction}\ t' \in \text{scope}(t), \text{it holds}\ \rho'_1(t') \subseteq \mathcal{L} \sigma \text{and} \rho'_2(t') \subseteq \mathcal{L} \sigma\). While instructions that belong to \(\text{scope}(t)\) are executed, since the annotation of values is upgraded to the level of the environment of the instructions, it holds: (i) if a variable is updated, then the stored value has annotation \(\mathcal{L} \mathcal{L} \sigma\); (ii) each item pushed onto the stack has annotation \(\mathcal{L} \mathcal{L} \sigma\); and (iii) no input or output channel \(a\) with \(\mathcal{P}(a) \subseteq \mathcal{L} \sigma\) is affected, otherwise \(\text{sem}\) would not respect the condition imposed by the theorem (since the environment of the instruction is \(\mathcal{L} \mathcal{L} \sigma\), if a channel is updated, the annotation of the channel would rise to a level \(\mathcal{L} \mathcal{L} \sigma\)). Two cases are possible:

**Case 1.** The instruction \(\hat{t} = ipd(t)\) does not belong to \(\text{scope}(t)\) (i.e. there is no loop). In this case both computations reach \(\hat{t}\). Let \(\hat{q}_1 = (\hat{t}, \hat{\rho}_1, \hat{\mu}_1, \hat{s}_1, \hat{c}_1)\) and \(\hat{q}_2 = (\hat{t}, \hat{\rho}_2, \hat{\mu}_2, \hat{s}_2, \hat{c}_2)\), respectively, the corresponding states. Note that \(\hat{\rho}_1 =_\sigma \hat{\rho}_2\). Let \(i\) and \(j\) be the minimum indices of the chain \(\text{sem}\), such that \(\hat{q}_1 \in \text{sem}_i\) and \(\hat{q}_2 \in \text{sem}_j\). Due to the lifting applied by \text{next}, there are two states in \(\text{sem}_{\text{max}(i,j)}(t), \hat{q}_1 = (\hat{t}, \hat{\rho}_1, \hat{\mu}_1, \hat{s}_1, \hat{c}_1)\) and \(\hat{q}_2 = (\hat{t}, \hat{\rho}_2, \hat{\mu}_2, \hat{s}_2, \hat{c}_2)\), corresponding, respectively, to the lifting of \(\hat{q}_1\) and \(\hat{q}_2\), i.e. with the same execution values, respectively, of \(\hat{q}_1\) and \(\hat{q}_2\), but with the security levels upgraded to the lub of all the states in \(\text{sem}_{\text{max}(i,j)}(t)\). Consider a variable \(x\). Due to the lifting operation issued by \text{next}, either both \(\hat{\mu}_1(x)\) and \(\hat{\mu}_2(x)\) have annotation \(\subseteq \mathcal{L} \sigma\), or both have annotation \(\mathcal{L} \mathcal{L} \sigma\). In the first case, \(x\) has not been affected by any instruction in \(\text{scope}(t)\) and thus \(\hat{\mu}_1(x) = \mu_1(x)\) and \(\hat{\mu}_2(x) = \mu_2(x)\). Since \(\mu_1 =_\sigma \mu_2\), then, by transitivity of equality, \(\hat{\mu}_1(x) =_\sigma \hat{\mu}_2(x)\). Since the reasoning holds for all variables, \(\hat{\mu}_1 =_\sigma \hat{\mu}_2\). A similar reasoning can be made for channels. Consider now the two stacks \(\hat{s}_1\) and \(\hat{s}_2\). They have the same length by the language assumption, since the corresponding instruction \(\hat{t}\) is the same. Consider a position \(1 \leq i \leq \hat{s}_1\); due to the lifting operation performed by \text{next}, either both \(\hat{s}_1[i]\) and \(\hat{s}_2[i]\) have annotation \(\subseteq \mathcal{L} \sigma\), or both have annotation \(\mathcal{L} \mathcal{L} \sigma\). Suppose \(\hat{s}_1[i] = (k_1, t_1)\) and \(\hat{s}_2[i] = (k_2, t_2)\) with \(t_1, t_2 \subseteq \mathcal{L} \sigma\). Note that \((k_1, t_1)\) and \((k_2, t_2)\) were already present in \(s_1\), respectively, \(s_2\) (the stacks immediately after the execution of instruction \(t\)), since any pushing operation issued inside \(\text{scope}(t)\) pushes a value with
higher annotation. Let \(s_1[j_1] = (k_1, \tau_1)\) and \(s_2[j_2] = (k_2, \tau_2)\). The stack portion from the bottom of \(s_1\) to \(i\) is equal to that from the bottom of \(s_1\) to \(j_1\), since \((k_1, \tau_1)\) was not popped out. Analogously, the stack portion from the bottom of \(s_2\) to \(i\) is equal to that from the bottom of \(s_2\) to \(j_2\). Since \(s_1\) and \(s_2\) have the same length, then \(j_1 = j_2\). By the equivalence of \(s_1\) and \(s_2\) it holds \(s_1[j_1] = V / E_{\text{sc}} s_2[j_1]\), and by transitivity \(s_1[i] = V / E_{\text{sc}} s_2[i]\). Since this occurs for each \(i, 1 \leq i \leq \#\hat{s}_1 = \#\hat{s}_2\), we have that \(\hat{s}_1 = s_2\). The above reasoning can be iterated following the two computations from, respectively, \(\hat{q}_1\) and \(\hat{q}_2\).

**Case 2.** All subsequent instructions of the program belong to \(\text{scope}(i)\) (there is a loop). By hypothesis, channels with security level \(\sqsubseteq \sigma\) cannot be affected from this point on by either of the two computations and thus the property is satisfied. \(\Box\)

5. Abstract domains

Abstract interpretation consists of a concrete and an abstract domain and two functions between them: an abstraction function \(\alpha\) and a concretization function \(\gamma\). The kind of abstraction and concretization function are chosen depending on the property to prove. Nevertheless, to ensure the correctness of the method, the two functions have to be related by a Galois connection or a Galois insertion [21]. Let us recall the definition of Galois insertion.

**Definition 5.1 (Galois insertion).** Let \((C, \sqsubseteq)\) and \((A, \sqsubseteq)\) be two complete lattices. Two functions \(\alpha: C \mapsto A\) and \(\gamma: A \mapsto C\) form a Galois insertion between \((C, \sqsubseteq)\) and \((A, \sqsubseteq)\), iff all the following conditions hold:

- \(\alpha\)-Monotonicity: \(\forall y, y' \in C. y \sqsubseteq y' \Rightarrow \alpha(y) \sqsubseteq \alpha(y')\)
- \(\gamma\)-Monotonicity: \(\forall a, a' \in A. a \sqsubseteq a' \Rightarrow \gamma(a) \subseteq \gamma(a')\)
- Galois: \(\forall y \in C. y \sqsubseteq \gamma(\alpha(y))\)
- Insertion: \(\forall a \in A. \alpha(\gamma(a)) = a\)

The abstract domains of our abstract interpretation are obtained by retaining only the security annotation of the data. Fig. 11 shows the abstract domain of data \(\mathcal{V}^\L\): abstract values are security levels plus an undefined value \(\bot\). The abstraction of a set of concrete values is \(\bot\) if the set is empty, otherwise it is the lub of the security levels of the annotations of the values in the set. The concretization of an abstract value is \(\emptyset\) for \(\bot\) and contains all concrete values that have an annotation less than or equal to \(\gamma(\alpha(a))\), for a level \(\sigma\).

An abstract memory maps every variable to an abstract value. Fig. 12 shows that the abstract memories form a lattice. The abstract domain \(\mathcal{C}^\L\) for channel is obtained from the concrete one simply by removing the execution values part, that is: \(\mathcal{C}^\L = \text{Names} \rightarrow \mathcal{L}\), and given \(y \in \varphi(\mathcal{C}), \forall a \in \text{Names}\),

\[
\alpha_C(y)(a) = \bigcup_{\mathcal{L}} \{\sigma \mid c(a) = (s, \sigma), c \in y\}
\]

<table>
<thead>
<tr>
<th>(\mathcal{V}^\L)</th>
<th>(\mathcal{L})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(\mathcal{L})</td>
<td>(\sigma)</td>
</tr>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\mathcal{L})</td>
</tr>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\mathcal{V}^\L)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Fig. 11. Domain of abstract values and abstraction/concretization functions.
Given $c^\natural \in C^\natural$ the concretization function $\gamma_C: C^\natural \rightarrow \wp(C)$ can be defined as:

$$\gamma_C(c^\natural) = \{ c \mid c \in C, \forall a \in \text{Names}, c(a) = (\delta, \sigma), \delta \in Z^*, \sigma \subseteq L, c^\natural(a) \}$$

**Proposition 5.2.**

(i) $\alpha_V$ and $\gamma_V$ form a Galois insertion between $V$ and $Y_V$.

(ii) $\alpha_M$ and $\gamma_M$ form a Galois insertion between $M$ and $M^\natural$.

(iii) $\alpha_C$ and $\gamma_C$ form a Galois insertion between $C$ and $C^\natural$.

**Proof.** By definition. □

It remains to describe how stacks are modeled in the abstract semantics. This is done in the following subsection.

### 5.1. Abstract stacks

A useful property of abstract interpretation is that the abstract domain is finite. This means that the approach can be used as the basis of a verification tool. We define a domain of abstract stacks which is parametric with respect to a maximum length $n$: thus a concrete stack of length $m \geq n$ has an approximated abstraction. This approximation helps to reduce the memory requirements of the verification tool.

Given a natural $n > 1$, the abstract domain $S^\natural_n$ of stacks with length at most $n$ consists of sequences of $n$ abstract values in $V^\natural$ having a (possibly empty) prefix of security levels, followed by a sequence of bottom values $\bot_V$. The length of the abstract stack is the prefix different from $\bot_V$. An empty stack is abstracted as a sequence of $n$ bottom values. For example, an abstract stack of length 2 in $S^\natural_4$ is $[\sigma_1, \sigma_2, \bot_V, \bot_V]$. The domain $S^\natural_n$ has also a bottom element $\bot_S$.

Consider a concrete stack

$$s = [(v_1, \sigma_1) \cdots (v_m, \sigma_m)]$$

If $m < n$, then its abstraction is the sequence of the $m$ security levels of the concrete items, followed by $n - m$ bottom values:

$$[\sigma_1 \cdots \sigma_m, \bot_V \cdots \bot_V]$$

If $m \geq n$, then the first $n - 1$ elements of the abstraction is the sequence of the security levels of the first $n - 1$ items, and the last element collapses the security levels of the remaining $m - n + 1$ concrete items:

$$[\sigma_1 \cdots \sigma_{n-1} \cdot \bigsqcup_{i=n-m}^{n-1} \sigma_i]$$
Clearly, in this case we lose some information on the elements from the \( m \)-th to the stack bottom. For example, the abstraction for \( n = 2 \) of \( s = [(v_1, \sigma_1) \cdot (v_2, \sigma_2) \cdot (v_3, \sigma_3)] \) is \([\sigma_1 \cdot (\sigma_2 \sqcup \sigma_3)]\).

The abstract operations on the stack \( \text{push}^\# : L \times S_n^\# \rightarrow S_n^\# \) and \( \text{pop}^\# : S_n^\# \rightarrow S_n^\# \) are defined as follows:

\[
\begin{align*}
\text{pop}^\#(\bot_S) &= \bot_S \\
\text{pop}^\#([v_1^\# \cdot \cdots \cdot v_n^\#]) &= [v_2^\# \cdot \cdots \cdot v_{n-1}^\# \cdot v_n^\#] \\
\text{push}^\#(\sigma, \bot_S) &= [\sigma \cdot \bot_S] \\
\text{push}^\#(\sigma, [v_1^\# \cdot \cdots \cdot v_n^\#]) &= [\sigma \cdot v_1^\# \cdot \cdots \cdot v_{n-2}^\# \cdot (v_{n-1}^\# \sqcup v_n^\#)]
\end{align*}
\]

The following proposition proves the monotonicity of \( \text{pop}^\# \) and \( \text{push}^\# \).

**Proposition 5.3.** \( \text{pop}^\# \) and \( \text{push}^\# \) are monotonic on \( S_n^\# \) for each \( n > 1 \).

**Proof.** See Appendix A. \( \square \)

Note that the pop operation, after removing the top element of the stack, duplicates the bottom element of the original stack. The push operation collapses the last two elements of the original stack to obtain the new bottom element. In this way the bottom element of the stack is always greater than or equal to each element that was lost due to a push operation on a full stack.

As an example, consider the domain \( S_2^\# \) of abstract stacks with maximum length \( n = 2 \). Fig. 14 shows this domain when the lattice of security levels contains only two levels: \( L = \{L, H\} \) with \( L \sqsubseteq H \). The complete domain with its abstraction and concretization functions is shown in Fig. 13.

The following proposition states some properties of the abstract data type stack. In particular, when some information is lost, the stack can be upgraded, but never downgraded.

![Fig. 13. Domain of abstract stacks with \( n = 2 \) and abstraction/concretization function.](image)

![Fig. 14. The lattice of abstract stacks \( (n = 2, L = \{L, H\}) \).](image)
Proposition 5.4. Given \( n > 1 \) and \( s^\natural \in S_n^\natural \),

(a) \( s^\natural \sqsubseteq S \) \( \operatorname{pop}^\natural (\operatorname{push}^\natural (\alpha, s^\natural)) \)
(b) If the length of \( s^\natural \) is less than \( n \), then: \( \operatorname{pop}^\natural (\operatorname{push}^\natural (\alpha, s^\natural)) = s^\natural \)
(c) \( \operatorname{push}^\natural (\alpha, \operatorname{pop}(\alpha^\prime \cdot s^\natural)) = [\alpha \cdot s^\natural] \)

Proof. By definition. \( \square \)

Point a means that issuing first a push operation and after a pop one, the resulting stack may be different from the original one, but always greater than or equal to it. Consider, for example, the domain \( S_n^\natural \) of abstract stacks and the stack \( s^\natural = \{L \cdot H\} \) with length 2: \( \operatorname{pop}^\natural (\operatorname{push}^\natural (L, \{L \cdot H\})) = \operatorname{pop}^\natural (\{L \cdot H\}) = [H \cdot H] \). Point b ensures that equality holds if the length of the abstract stack is strictly less than \( n \). Point (c) means that in the case of a pop followed by a push no information about the bottom part of the stack is lost in any case.

The following proposition will be useful for proving the correctness of the abstract semantics.

Proposition 5.5. Let \( s \in S \) be a concrete stack.

(a) \( \alpha_S (\operatorname{push}(\gamma, \tau, s)) = \operatorname{push}^\natural (\tau, \alpha_S (s)) \)
(b) \( \alpha_S (\operatorname{pop}(s)) \sqsubseteq S \) \( \operatorname{pop}^\natural (\alpha_S (s)) \)
(c) If \( n > s \), then \( \alpha_S (\operatorname{pop}(s)) = \operatorname{pop}^\natural (\alpha_S (s)) \)

Proof. See Appendix A. \( \square \)

The above proposition states that, while the order in which \( \operatorname{push}^\natural \) and \( \alpha_S \) are performed is permutable, if we first apply \( \operatorname{pop} \) and then we abstract the resulting stack, we obtain a result which may be less than the inverted application of \( \operatorname{pop}^\natural \) and \( \alpha_S \) (point b). Point c ensures that equality holds if the concrete stack has length less than \( n \). As an example, consider \( n = 2 \) and the stack \( s = [(k_1, \alpha_1) \cdot (k_2, \alpha_2)] \) with length 2. We have

\[
\alpha_S (\operatorname{pop}(s)) = \alpha_S ((k_2, \alpha_2)) = [\alpha_2 \cdot \bot_V]
\]

while

\[
\operatorname{pop}^\natural (\alpha_S (s)) = \operatorname{pop}^\natural ([\alpha_1 \cdot \alpha_2]) = [\alpha_2 \sqcup \alpha_2]
\]

Finally, the following proposition states the correctness of the abstraction on stacks.

Proposition 5.6. Functions \( \alpha_S \) and \( \gamma_S \) form a Galois insertion between \( S \) and \( S_n^\natural \) for each \( n > 1 \).

Proof. See Appendix A. \( \square \)

From now on, we omit the subscript \( n \) from \( S_n^\natural \) when it is not relevant.

5.2. Abstract states

The abstract domain of states is \( Q^\natural : B \to (L \times M^\natural \times S^\natural \times C^\natural) \). It contains all the functions that associate the instruction addresses with elements in \( (L \times M^\natural \times S^\natural \times C^\natural) \). Given an abstract state \( q^\natural \in Q^\natural \), and an instruction label \( t \in B \), \( q^\natural (t) = \{\alpha, \mu^\natural, s^\natural, c^\natural\} \) is a tuple made up of a security level representing the security environment of \( t \), the abstract memory, stack and channels.

Given \( q^\natural \in Q^\natural \), we use \( q^\natural (.\text{env}) \), \( q^\natural (.\text{mem}) \), \( q^\natural (.\text{stack}) \) and \( q^\natural (.\text{chan}) \) to denote \( \alpha, \mu^\natural, s^\natural, c^\natural \), respectively. We also use the notation \( q^\natural (t) \cdot [f_1, f_2, \ldots, f_n] \) to denote the fields \( \{f_1, f_2, \ldots, f_n\} \) of \( q^\natural (t) \) (for example, \( q^\natural (t) \cdot [\text{mem}, \text{stack}] \)). We denote by \( \text{dom}(q^\natural) = \{t | q^\natural (t) \cdot [\text{mem}, \text{stack}, \text{chan}] \neq \bot_{L \times M^\natural \times S^\natural \times C^\natural}\} \) the instruction addresses to which \( q^\natural \) assigns a defined value for memory, stack and channels. The abstract states are tuples consisting of \( n \) elements, where \( n \) is the number of instructions of the program. We have that \( (Q^\natural, \sqsubseteq, \sqcup, \cap, \sqcap, \bot, \top) \) is a lattice, where, as illustrated in Fig. 15, the operations are defined as the pointwise application of the corresponding operations on the fields of the abstract states.
Let us now consider the abstraction and concretization functions between the concrete and abstract domains of the states.

The abstraction function \( /VTQ : \wp(Q) \rightarrow Q^\# \) is defined as follows. Let \( Q \) be a set of concrete states in \( Q = B \times Env \times M \times S \times C \). We define \( /VTQ(Q)(t) = \langle /ESC, /SYN^\#, s^\#, c^\# \rangle \) where

\[
/ESC = \bigsqcup L \{ /SUB(t') | \langle t', /SUB, /SYN, s, c \rangle \in Q \}
\]

\[
/SYN^\# = /VTM(\{ /SYN | \langle t, /SUB, /SYN, s, c \rangle \in Q \})
\]

\[
s^\# = /VTS(\{ s | \langle t, /SUB, /SYN, s, c \rangle \in Q \})
\]

\[
c^\# = /VTC(\{ c | \langle t, /SUB, /SYN, s, c \rangle \in Q \})
\]

Note that the environment of \( t \) is calculated by considering the environments that belong to all the states in \( Q \), while the memory, the stack and the channels are taken only from the states in \( Q \) that have \( t \) as program counter. In fact the machine state is local information, while the environment is global information.

Moreover, if an instruction \( t \) does not occur in \( Q \), then the set of memories, channels and stacks associated with it are empty. Thus the abstraction functions \( /VTM, /VTS \) and \( /VTC \) will produce bottom values of the respective domains, excluding \( t \) from \( \dom(/VTQ(Q)) \).

For the concretization function we have:

\[
yQ(q^\#) = \{ \langle t, \rho, \mu, s, c \rangle | t \in \dom(q^\#), \forall t' \in B, \rho(t') \sqsubseteq q^\#(t').env,\mu, s, c \in yMxSxC(q^\#(t), [mem, stack, chan]) \}
\]

The following theorem prove the correctness of the abstraction on states.

**Theorem 5.7.** Functions \( \alphaQ \) and \( \gammaQ \) form a Galois insertion between \( Q \) and \( Q^\# \).

**Proof.** See Appendix A. \( \square \)

### 6. Abstract semantics and correctness

In this section, we give an abstract semantics that allows us to finitely execute the program in the abstract domain. The rules of the abstract semantics are shown in Fig. 16. If the premise of a rule is true, the state \( q^\# \) is transformed in the way described by the rule. Given a domain \( D \) of tuples and a variable \( d \) with values in \( D \), the notation \( d \sqcup d' = d' \) means \( d = d \sqcup d' \). For the sake of simplicity, in the conclusion of the rule only the part of \( q^\# \) that is updated is indicated. For example, in the rule for \( pop \), the only elements to be changed in the new state are the fields \( [mem, stack, chan] \) of \( q^\#(t + 1) \), and the new value of these fields is the lub between the old value and the pair \( [\mu^2, pop^2(s^2), c^2] \). Each rule behaves like the corresponding in the concrete semantics, but acts on security levels instead of execution data. It modifies the fields \( mem, stack \) and \( chan \) of the immediate successor instructions: \( t + 1 \) for all non-jumping instructions, \( j \) for unconditional jumps, \( t + 1 \) and \( j \) for conditional jumps.

Rule \( if \) also modifies the field \( env \) of all the instructions in \( \text{scope}(t) \). We assume that the transition can only be applied to well-formed states, i.e. only to states with non-bottom values for memory, stack and channels.
\[ q^t(t + 1).[\text{mem}, \text{stack}, \text{chan}]_{\mathcal{L} \in \mathcal{C}} = \begin{cases} \mu^t, & \text{if } t = 0 \\ \mu^t, \text{push}^t(\tau) \cup \mu^t \cup \mu^t(\text{pop}^t(s^t)) & \text{otherwise} \end{cases} \]

**Definition 6.1** (initial abstract state). Let \( P \) be a program and \( \mathcal{P} \) a security policy for \( P \). The initial state \( q^0 \) of the abstract semantics is defined as:

\[ q^0(t) = \langle \mu^0, s^0, c^0 \rangle \]

**Definition 6.2** (next operator). Given an abstract state \( q^t \), the application of the next operator yields the state reached in one step of computation from each instruction:

\[ \text{op: } q^t(t) = \langle \mu^t, [\tau \cdot \sigma \cdot v_1 \cdots v_{n-1}] \rangle \]

\[ \text{pop: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{push: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{load: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{store: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{send: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{receive: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{goto: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{if: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

\[ \text{halt: } q^t(t) = \langle \mu^t, s^t, c^t \rangle \]

Fig. 16. Abstract semantics.

Let us denote by \( \rightarrow \) the relation between the abstract states induced by the rules.

**Definition 6.1** (initial abstract state). Let \( P \) be a program and \( \mathcal{P} \) a security policy for \( P \). The initial state \( q^0 \) of the abstract semantics is defined as:

\[ q^0(t) = \langle \mu^0, \mu^0_0, \mu^0_1, \cdots, \mu^0_{n-1} \rangle \]

**Definition 6.2** (next operator). Given an abstract state \( q^t \), the application of the next operator yields the state reached in one step of computation from each instruction:

\[ \text{next}^t(q^t) = \bigcup_{q' \in \mathcal{Q}} \{ q' | q' \rightarrow q^t \} \]

**Proposition 6.3** (monotonicity of next^t). next^t is monotonic in \( (\mathcal{Q}^t, \subseteq) \).

**Proof.** By the abstract semantic rules. \( \square \)
**Definition 6.4 (abstract semantics).** The abstract semantics $\text{sem}^\flat \in Q^\flat$ is the least upper bound in $(Q^\flat, \sqsubseteq_Q)$ of the following increasing chain, defined for all $n \in \mathbb{N}$:

\[
\begin{align*}
\text{sem}^\flat_0 &= q^\flat_0 \\
\text{sem}^\flat_{n+1} &= \text{next}^\flat(\text{sem}^\flat_n)
\end{align*}
\]

The following theorem states the correctness of the abstract semantics.

**Theorem 6.5 (Global correctness).** $\alpha_Q(\text{sem}) \sqsubseteq \text{sem}^\flat$.

**Proof.** See Appendix A. □

**Corollary 6.6.** A program $P$ has secure information flow under a security policy $\mathcal{P}$, if, for each $t \in B$, the following condition holds: if $\text{sem}^\flat(t) = \langle t, \mu^\flat, s^\flat, c^\flat \rangle$, then $\forall a \in \text{Name}\text{s}, c(a) \sqsubseteq_L P(a)$.

**Proof.** Combining Theorem 6.5 with Theorem 4.8. □

Note that the security level of the output channels produced by the instruction level security analysis allows us to identify secure channels with respect to $\mathcal{P}$, independently from the safety of the program. In fact a channel $a$ such that $\text{sem}^\flat.chan(a) \sqsubseteq_L P(a)$ can be considered secure, since the information flowing through $a$ is not affected by data with a level higher than $P(a)$.

The following proposition connects the accuracy of the analysis with the size of abstract stacks.

**Proposition 6.7.** Let us denote by $\text{sem}^\flat_n$ the abstract semantics in the case of abstract stacks of size $n$ (that is the domain of the abstract stacks is $S^\flat_n$). Then, $\forall n \geq 1$, $\forall t \in B$

1. $\text{sem}^{\flat,n+1}(t).chan \sqsubseteq_C \text{sem}^{\flat,n}(t).chan$
2. if $m$ is the maximum length of the concrete stack in any execution, $\forall n > m$: $\text{sem}^{\flat,n+1}(t).chan = \text{sem}^{\flat,n}(t).chan$.

**Proof.** From Proposition 5.4. □

Point a states that greater is the size of the abstract stack, more precise is our analysis. Point b claims that, to achieve the maximum precision, it suffices to use in the analysis abstract stacks of size $n + 1$, if $n$ is the maximum length of the execution stack.

**Example.** To see an example of how the precision of the analysis increases with the size of the abstract stack, look at Fig. 17. The second and the third columns show the abstract states calculated by the abstract semantics with $n = 2$ and $n = 3$, respectively. At instruction 8, with $S^\flat_2$, the top of the stack (the item to be sent) is $H$ and hence the level of the public output channel $\text{pubO}$ becomes $H$, whereas with $S^\flat_3$, the top of the stack is $L$ and $\text{pubO}$ maintains $L$ as level. Hence the program is accepted with $S^\flat_3$, but not with $S^\flat_2$. Increasing further the size of the abstract stacks does not result in any improvement, since the maximum length of the execution stack is 2.

A slight modification of the method can be used when the security policy assigns a level only to a subset of the channels. To perform this analysis, in the initial state only channels belonging to the domain of the policy are set to their level, while all other input and output channels are set to the minimum level. In this case the level computed for the not fixed channels is an output of the checker: security can be ensured by not allowing users that have a certain confidentiality level to access channels for which a higher level was calculated by the analysis. This calculation of the level of channels may enable code to be reused, especially when more complex security lattices are used.

The analysis can also be used if we connect programs by means of the channels to obtain a concurrent program. In this scenario, a policy assigns a level only to the external channels, but not to channels connecting
two processes. The level of these channels may be computed by iteratively applying the analysis to the processes, starting from the assignment to internal channels of the minimum level, until a fixpoint assignment to these channels is reached.

6.1. Notes for an implementation

The abstract semantics is the basis of the instruction-level security analysis. The abstract rules can be applied until a fixpoint is reached. A static checker can perform this calculation using for instance the Kildall working list algorithm [35] (as the Java Bytecode Verifier does).

Let us now briefly discuss the complexity of the analysis. The space complexity is $O(N \cdot \log(M) \cdot n)$ where $N$ is the total number of variables and channels plus the size of the abstract stack, $M$ is the number of elements in $L$, and $n$ is the number of instructions. The time complexity is theoretically $O(N^2 \cdot M \cdot n)$. In fact, every application of an abstract rule has a linear complexity in $N$ due to the least upper bound operation on the abstract memory, and, in the worst case, the abstract state of every instruction can have up to $O(N \cdot M)$ different values during the verification process. However, in practice, the number of abstract executions is much smaller. As suggested by Leroy [38], the analysis can be conducted at the level of the basic blocks instead of single instructions, saving only the state for the beginning of each basic block and calculating the others on the fly. This helps to reduce the space complexity to $O(N \cdot \log(M) \cdot B)$, and the time complexity to $O(N^2 \cdot M \cdot B)$, where $B$ is the number of basic blocks.

7. A non-trivial example

In this section, we present the analysis of a simple yet expressive example. Imagine we have an application that provides information about tax rates (see Fig. 18(c)). Let us suppose there are two possible tax rates: one for incomes below a threshold, and the other for higher incomes. The rates and the threshold are not hard-coded in the program, but are retrieved from the network, since their values can change. Users can request:

- the specific rate they have to apply to their own income (request code 0), by providing the income;
- the value of the threshold (request code 1);
- the value of the low rate (request code 2);
- the value of the high rate (request code 3).

---

Fig. 17. The size of the abstract stack affects the accuracy of the analysis.
In order not to overload the network, the application caches the threshold and the rates, and only updates their values if there is a code 0 request. The application fetches the values of the threshold and the rates from the corresponding input channels (threshold, lowRate and highRate). Users communicate with the application by putting the code request into the channel userReq. The application retrieves firstly the threshold and the rates from the network, and then loops forever, accepting and serving users requests. When there is a code 0 request, since incomes and calculated tax rates are assumed to be confidential, the application uses the dedicated pair of channels income and rateRes, while responses to other requests are sent through the channel userRes.²

Therefore a possible security policy could be:

\[ \mathcal{P}(\text{threshold}) = \mathcal{P}(\text{lowRate}) = \mathcal{P}(\text{highRate}) = L \]
\[ \mathcal{P}(\text{clientReq}) = \mathcal{P}(\text{clientRes}) = L \]
\[ \mathcal{P}(\text{income}) = \mathcal{P}(\text{rateRes}) = H \]

A first version of the application can be seen in Fig. 18(a) (source code in 18(b)). The opcode eq (geq) pops the two topmost items off the stack and pushes 1 if the first is equal to (or greater than) the second, 0 otherwise.

We can observe that, under this policy, the program does not respect the secure information flow property. In fact, an attacker that is able to observe the channel lowRate and highRate can gather information about the incomes by observing which of the two tax rates are requested to the network. This conjecture can be verified by computing the abstract semantics of the program with abstract stacks of length 2 (see Fig. 19). In fact, the security level of lowRate, highRate and userRes exceeds that specified by the policy.

We can correct the program by changing the case ′0′ branch, as shown in Fig. 20. In this version, the application requests both rates to the net, whatever the value of the income is, and then only sends the proper rate to the clientRes channel. The abstract semantics is in Fig. 21. The table highlights that the levels of the channels are always equal to those specified by the policy \( \mathcal{P} \). The program can thus be considered secure. Note that the variable \( \text{tmp} \) holds both confidential data (the income, with level \( H \)) and public data (the code of the request, with level \( L \)), but in different instructions of the program: at instructions 7–8, and 38–46 the level \( \text{tmp} \) is \( H \), in all the other instructions it is \( L \). The ability to have security annotations for the same entity which are instruction-dependent makes our method flow-sensitive, as discussed below.

8. Related work

The secure information flow property of programs was pioneered by a number of authors: Bell and La Padula [12], Denning [25], and Goguen and Meseguer [29]. Denning and Denning [26], Andrews and Reitman [4], and subsequently Banâtre et al. [6], addressed program certification, which statically analyses secure information flow.

Most recent work that presents methods to check secure information flow is based on security typing: each variable is associated with a security type and secure information flow is checked by means of a type system [1, 11, 30, 36, 41, 43, 44], see also the exhaustive survey presented by Sabelfeld and Myers [42]. A type-based approach was also described in previous papers of Zdancewic and Myers [46, 47], where non-interference is studied in imperative higher-order languages by means of a continuation passing style (CPS) translation of the programs. Linear continuations are used to handle implicit flows. The advantage of these approaches is efficiency, since fast typing algorithms can be used. On the other hand, most of these methods are flow-insensitive, since the analysis is independent of the order in which the instructions are executed. Consider again the lattice \( \mathcal{L} = \{L, H\}, L \sqsubseteq H \).

The simple high program fragment \( l=h; \ 1=0; \) where \( h \) holds high-level information and \( l \) is required to hold only public data, is secure if we observe the final state, where the contents of variable \( l \) is 0, irrespectively, of the initial contents of variable \( h \). On the other hand, the program \( l=0; \ l=h; \), where the order of the two

² For instance, income and rateRes could refer to a SSL connection, while the other channels to plain-text connections.
instructions is inverted, is not safe, since the final value of \( l \) depends on the initial value of \( h \). A fixed security typing approach rejects both programs, since it computes for \( l \) the type (level) \( H \). In fact, in fixed security typing the same security type is computed for each variable and holds for the whole program. Hence, the assignment \( l = h; \) is considered not correct, whenever it occurs in the program, since \( h \) has type \( H \) and \( l \) has type \( L \).
An approach is called flow-sensitive if the corresponding analysis provides a different abstraction for variables at different points of the program. A flow-sensitive approach distinguishes the two programs above, and certifies the first, but not the second. In fact, for the first program, at the first instruction the security type (level) of $l$ is $H$, but at the second one it is $L$. Hence the program can be certified (obviously if the safety requirements concern only the final state).
Many flow-sensitive methods have been defined to check secure information flows. Some of them are based on data-flow analysis [19, 27]. Both compute the data/control-flow dependencies among the variables of the program. Other works [34, 3, 32] are based on logics. Joshi and Leino [34] use a Hoare-like logics, while the approach by Jacobs et al. [32] is based on the PVS theorem prover. Amtoft et al. defined in a recent paper [3] an inter-procedural and flow-sensitive analysis for a Java-like language. Their approach exploits assertions in the logic that can be either provided by the data and the control flow analysis, or specified directly by the programmer (à la JML). The data/control-flow based and logical approaches to information flow generally allow a finer inspection of the properties than a typical security typing approach, but they may be less efficient.

Our analysis is flow-sensitive: it allows a memory location, stack element, or channel to hold data with a different secrecy level in different instructions of the program. Hence it is more precise than fixed security typing and is able to certify a larger class of programs such as, for example, the equivalent in our language of

```java
l = h;
l = 0;
```

A recent paper by Hunt and Sands [31] introduces a flow-sensitive security typing system, that is able to calculate the security type of the variables of a simple While language in a context-sensitive way. It also shows that the powerset of the variables is a universal lattice of types from which all other typings can be obtained. If we compare this approach to ours, and allow for the fact that different languages, definitions and formalisms are considered in the two works, the two methods would seem to achieve the same level of accuracy. Hunt and Sands also present an algorithm that transforms a program that can be typed in the flow-sensitive typing system into one that can be typed in a flow-insensitive typing system. The transformation adds a number of variables to the original program. A similar transformation for Java bytecode was suggested by Leroy [37] to reduce the memory space for the bytecode verifier when it is installed on smart-cards. The algorithm adds new variables and makes some other transformations in order to manage a unique state instead of a table with a state configuration for each instruction. This algorithm can be simply adapted to reduce the memory need for instruction-level security analysis.
Other recent works have analysed information flow in the framework of abstract interpretation [45,28]. The method proposed by Zanotti [45] is different from our approach since in their case the abstract domain is the lattice of types. As a consequence, the method is flow-insensitive and has the power of fixed security typing. The focus of the paper by Giacobazzi and Mastroeni [28] is the notion of abstract non-interference. They observe that classical non-interference does not allow private information to be disclosed at all, thus being a property that is

Fig. 21. The tax example: abstract semantics of the secure version.
too strict for use in practical contexts. In many cases, the attacker cannot observe all the information sent to the output, but only some of its properties. On the other hand, if the attacker is modeled as an abstract interpreter, programs that leak private information can be deemed as being secure, provided that such information is not observable by the attacker. In this approach the abstract interpretation is exploited to model observation instead of computation.

The SLam calculus of Heintze and Riecke [30] is a typed λ-calculus, that maintains security information as well as type information. Data are annotated with security types. The type system propagates both secrecy and integrity, maintaining distinct forms of security information: for example, which agents could have created an object or which agents may read an object. A finer grained control of security is achieved, due to the richness of the domain of types and the fact that different levels of security may be assigned to the different components of an object. The language has been extended also to include commands of imperative languages, such as assignments. It could be interesting to define an abstract interpretation acting on the domain of security levels used in this work.

Finally, we remark that our definition of secure information flow property does not concern only the final state of the program, as in nearly all the approaches cited above, but also the intermediate states of the execution, and in particular the whole sequence of manipulations of the input and output channels during the execution. In most approaches, instead, they are the variables to be associated with a security level and their values in the final state is observed. Hence, there is a need to distinguish between a weak version of the property, which assumes termination, and a strong version, that also verifies the possible non-termination of the program due to an incorrect information flow. Our formulation of the property does not depend on the possible non-termination of the program. In fact, even a program that may not terminate due to an incorrect flow can be σ-secure if the contents of the channels with a security level less than or equal to σ is not affected by information with a level higher than σ.

8.1. Related work on secure information flow in low level languages

In 2002, Kobayashi and Shirane [36] adapted the Volpano [44] type system to low level languages. They analysed a small subset of the Java Virtual Machine Language. Their typing rules generate a set of constraints, which, once they have been reduced and solved, indicate whether the method respects non-interference. Their analysis is simpler than ours, since local variables types are not allowed to change, and thus the method is flow-insensitive. However, the proposed type inference algorithm has a time complexity of \(O(n^2)\) if \(n\) is the number of instructions, which is comparable to ours (see Section 6.1).

A recent work by Barthe and Rezk [11] describes a type system à la Volpano for checking classical non-interference in Java bytecode. Many of the advanced features of the language are addressed (objects, methods and exceptions). Implicit flows are handled by means of the concept of control dependencies region, similar to the \textit{scope} notion. Each method of a class can be verified separately from the others, according to a signature that assigns security types to parameters, object fields, return value and to the effect of the method on the heap. However, this analysis is not flow-sensitive either, since the security level of each variable is fixed throughout all the method. This can be a problem, since optimizing compilers may decide to reuse a variable for different purposes (i.e. to store confidential and public data in different portions of the method).

Moreover, both these papers [11,36] are less precise than our approach in relation to the stack. In fact, on entering a high \(\mathtt{if}\), they raise all the stack items to the \(\mathtt{H}\) value, while we do not modify the stack. Consider the simple program in Fig. 22(a) (a possible source code is in Fig. 22(b)) and suppose that \(x\) contains a private value. Since the guard of the \(\mathtt{if}\) is high, at instruction 3 the whole stack (constant value 1 included) is lifted to the high security level even though its bottom part is not touched by the computation. In our approach, instead, the stack is left unchanged, thus eventually resulting in a low value in the variable \(z\). Our choice of not lifting the stack at the conditionals is still safe since:

- every time a value is pushed into (loaded from) the stack, this value is upgraded with the security level of the environment;
- the stack height is fixed for every instruction.
Therefore, we believe that this improvement is significant since, in assembler code, the stack is often used to temporarily store values that are then reused after a computation.

Medel et al. [40] proposed a flow-sensitive non-interference analysis for an assembly language with heap and registers. They extended a typed assembly language with two pseudo instructions that handle implicit flows. Every time a conditional instruction is reached, the label of the ipd is pushed into a stack of labels. When the ipd is reached, its label is popped from the stack and the security level of the program counter can be lowered. On the contrary, we calculate this information in advance, by computing the function scope.

A data-flow approach for low-level languages has been suggested by Genaim and Spoto [27]. Their analysis reveals all the flows (implicit and explicit) that may arise in Java Bytecode. All the object-oriented features of Java Bytecode are addressed (fields, methods, exceptions) have been addressed. The result of the analysis is a set of dependencies, represented by Boolean functions. The power of the analysis is comparable with that of our method. They provided an implementation that exploits binary decision diagrams for handling Boolean functions, in order to improve efficiency and reduce memory space.

8.2. Previous work of the authors

Some of our previous work have dealt with the topic of this paper. We have considered non-interference in timed automata [24] and parallel languages [10]; in this latter work, the secure flow is checked essentially by model checking. In another paper [15] we defined an approach to check secure information flow in Java bytecode, based on program transformation.

An abstract interpretation framework was used as the basis of a flow-sensitive analysis to check secure information flow in stack-based machine languages [13,8,9,14], while a similar method [7] was defined for high-level languages. All these papers consider a different security model of confidentiality, concerning only the final states of the program and not including channels. They define a set of abstract rules, by means of which an abstract transition system can be built. The analysis is more accurate than the one presented in the present paper. In fact, the same instruction may occur in different states of the transition system, and each state may be characterized by a different configuration of the memory and the stack. Consider, for example, the program fragment “

```java
1 push 1
2 load x
3 if 6
4 push 0
5 store y
6 store z
7 halt
```

Therefore, we believe that this improvement is significant since, in assembler code, the stack is often used to temporarily store values that are then reused after a computation.

Medel et al. [40] proposed a flow-sensitive non-interference analysis for an assembly language with heap and registers. They extended a typed assembly language with two pseudo instructions that handle implicit flows. Every time a conditional instruction is reached, the label of the ipd is pushed into a stack of labels. When the ipd is reached, its label is popped from the stack and the security level of the program counter can be lowered. On the contrary, we calculate this information in advance, by computing the function scope.

A data-flow approach for low-level languages has been suggested by Genaim and Spoto [27]. Their analysis reveals all the flows (implicit and explicit) that may arise in Java Bytecode. All the object-oriented features of Java Bytecode are addressed (fields, methods, exceptions) have been addressed. The result of the analysis is a set of dependencies, represented by Boolean functions. The power of the analysis is comparable with that of our method. They provided an implementation that exploits binary decision diagrams for handling Boolean functions, in order to improve efficiency and reduce memory space.

8.2. Previous work of the authors

Some of our previous work have dealt with the topic of this paper. We have considered non-interference in timed automata [24] and parallel languages [10]; in this latter work, the secure flow is checked essentially by model checking. In another paper [15] we defined an approach to check secure information flow in Java bytecode, based on program transformation.

An abstract interpretation framework was used as the basis of a flow-sensitive analysis to check secure information flow in stack-based machine languages [13,8,9,14], while a similar method [7] was defined for high-level languages. All these papers consider a different security model of confidentiality, concerning only the final states of the program and not including channels. They define a set of abstract rules, by means of which an abstract transition system can be built. The analysis is more accurate than the one presented in the present paper. In fact, the same instruction may occur in different states of the transition system, and each state may be characterized by a different configuration of the memory and the stack. Consider, for example, the program fragment “

```java
1: if (not l) l=h; else skip; 2: ...
```

where l is low and h is high. If l is not equal to 0, its value is not updated and therefore there is no incorrect information flow for l. On the other hand, if l=0, the final value of l is high and this violates the secure flow property. The abstract transition system contains two states for instruction 2: one state corresponds to the execution of the true branch of the conditional, in which l is high; the other state corresponds to the execution of the false branch, in which l is low. Hence it is possible to distinguish between these two situations by analyzing the transition system. If we apply the analysis described in the present paper, the program is not certified, since l is high in the (only) row corresponding to instruction 2. In fact the row contains the lub of the levels of all possible information flows affecting l when instruction 2 is executed. Unfortunately, because the number of states of the abstract transition system is too high, this abstraction is not suitable to be directly used as the basis of a tool for checking secure information flow. In fact other techniques would need to be combined with abstract interpretation: in the above papers we
used model checking to complete the verification process. A similar combination of abstract interpretation and model checking was suggested by Bibier et al. [17]. The method presented in the present work is much more efficient in time and space with respect to these previous approaches. This result is obtained by the definition of a different abstraction which is more concise, even though we loose some precision.

Acknowledgments

The authors thank the anonymous referees for their valuable comments. This work was partially supported by the Italian COFIN2004 Project “Aida: Abstract Interpretation Design and Applications”.

Appendix A. Proof of some propositions and theorems

A.1. Proof of Proposition 5.3

**Proposition 5.3.** \(\text{pop}^n\) and \(\text{push}^n\) are monotonic in \(S^n_2\) for each \(n > 1\).

**Proof.** The proof is shown only for \(n = 2\).

(\(\text{pop}^n\)) We want to prove that \(s^1_2 \in S \Rightarrow \text{pop}^n(s^1_2) \subseteq S \text{pop}^n(s^1_2)\). Consider \(s^1_2 = [\sigma_1 \cdot \tau_1] \land s^2_2 = [\sigma_2 \cdot \tau_2]\) (the trivial case \(s^1_2 = \bot\) is omitted). Then, since \(\tau_1 \subseteq \tau_2\) is true by hypothesis, \(\text{pop}^n(s^1_2) = [\tau_1 \cdot \tau_1] \subseteq S [\tau_2 \cdot \tau_2] = \text{pop}^n(s^1_2)\).

(\(\text{push}^n\)) We want to prove that \(\sigma_1 \subseteq \tau_1 \land s^1_2 \subseteq S \Rightarrow \text{push}^n(\sigma_1, s^1_2) \subseteq S \text{push}^n(\tau_1, s^2_2)\). We proceed by enumeration on \(s^1_2\) and \(s^2_2\):

\[
\begin{align*}
(s^1_2 = \bot \land s^2_2) & \Rightarrow \text{push}^n(\sigma_1, \bot) = [\sigma_1, \bot] \subseteq S [\tau_1, \bot] = \text{push}^n(\tau_1, s^2_2) \\
(s^1_2 = \bot \land s^2_2 = [\tau_2 \cdot \tau_3]) & \Rightarrow \text{push}^n(\sigma_1, \bot) = [\sigma_1, \bot] \subseteq S [\tau_1 \cdot \tau_2 \cup \tau_3] = \text{push}^n(\tau_1, s^2_2) \text{ since, by hypothesis, } \sigma_1 \subseteq \tau_1. \\
(s^1_2 = [\sigma_2 \cdot \sigma_3] \land s^2_2 = [\tau_2 \cdot \tau_3]) & \Rightarrow \text{push}^n(\sigma_1, s^1_2) = [\sigma_1 \cdot \sigma_2 \cup \tau_3] \subseteq S [\tau_1 \cdot \tau_2 \cup \tau_3] = \text{push}^n(\tau_1, s^2_2) \text{ since, by hypothesis, it holds that } \sigma_i \subseteq \tau_i, \ i \in \{1, 2, 3\}. \quad \square
\end{align*}
\]

A.2. Proof of Proposition 5.5

**Proposition 5.5.** Let \(s \in S\) be a concrete stack.

(a) \(\alpha_S(\text{push}(k, \tau, s)) = \text{push}^2(\tau, \alpha_S(s))\)

(b) \(\alpha_S(\text{pop}(s)) \subseteq S \alpha_S(\text{pop}^2(\alpha_S(s))\)

(c) If \(\#s < n\), then \(\alpha_S(\text{pop}(s)) = \text{pop}^n(\alpha_S(s))\)

**Proof.** Again, only the case \(n = 2\) is considered (\(S^n_2\) is the domain of the abstract stacks)

(a) We proceed by enumeration on \(s\):

\[
\begin{align*}
(s = \lambda) & \Rightarrow \alpha_S(\text{push}(k, \tau, \lambda)) = \alpha_S([(k, \tau)]) = [\tau, \bot] \\
(s = [(k_1, \tau_1)]) & \Rightarrow \alpha_S(\text{push}(k, \tau, [(k_1, \tau_1)]) = \alpha_S([(k, \tau) \cdot (k_1, \tau_1)]) = [\tau, \tau_1] \\
(s = [(k_1, \tau_1) \cdots (k_m, \tau_m)]) & \Rightarrow \alpha_S(\text{push}(k, \tau, s)) = \alpha_S([(k, \tau) \cdot (k_1, \tau_1) \cdots (k_m, \tau_m)]) = [\tau, \bigcup_{i=1}^{m} \tau_i] \\
(s = [(k_1, \tau_1) \cdots (k_m, \tau_m)]) & \Rightarrow \text{push}^2(\tau, \alpha_S(s)) = \text{push}^2([\tau, \bigcup_{i=2}^{m} \tau_i]) = [\tau, \tau_1 \cup \bigcup_{i=2}^{m} \tau_i] = [\tau, \bigcup_{i=1}^{m} \tau_i]
\end{align*}
\]
(b) Again, three cases depending on the length of \( s \) (the case where \( s = \lambda \) is omitted):

\[
\begin{align*}
(s &= [(k_1, t_1)]) & \alpha_S(pop(s)) &= \alpha_S(\lambda) = \bot_S \\
(s &= [(k_1, t_1) \cdot (k_2, t_2)]) & \alpha_S(pop(s)) &= \alpha_S([(k_2, t_2)]) = [t_2, \bot_V] \subseteq_S [t_2, t_2] = pop^v([t_1, t_2]) = pop^v(\alpha_S(s)) \\
(s &= [(k_1, t_1) \cdot \cdots \cdot (k_m, t_m)]) & \alpha_S(pop(s)) &= \alpha_S([(k_2, t_2) \cdot \cdots \cdot (k_m, t_m)]) = [t_2, \bigcup_{i=3}^m t_i] \subseteq_S [\bigcup_{i=3}^m t_i \cdot \bigcup_{i=2}^m t_i] = pop^v([t_1, \bigcup_{i=2}^m t_i]) = pop^v(\alpha_S(s)) \text{ where the inequality is ensured by the monotonicity of the operator } \sqcup.
\end{align*}
\]

(c) See the previous case. \( \square \)

A.3. Proof of Proposition 5.6

**Proposition 5.6.** \( \alpha_S \) and \( \gamma_S \) form a Galois insertion on \( S^\perp_\delta \) for each \( n \geq 1 \).

**Proof.** Again, the proof for the case \( n = 2 \) follows.

(\( \alpha \)-Monotonicity) Let \( y, y' \in r(\mathcal{S}) \). Since \( \alpha_V(y) = \bigcup_{s_i \in y} \alpha_S^1(s_i) \) and \( y \subseteq y' \), we can write \( \alpha_S(y') = \alpha_S(y) \sqcup S \bigcup_{s_i \in y'} \alpha_S^1(s_i) \). The monotonicity of the lub \( \sqcup \) thus ensures that \( \alpha_S(y) \subseteq S \alpha_S(y') \).

(\( \gamma \)-Monotonicity) We have to prove that \( \forall s_1^\perp, s_2^\perp \in S^\perp_\delta, s_2^\perp \subseteq_S s_1^\perp \implies \gamma_S(s_1^\perp) \subseteq \gamma_S(s_2^\perp) \). This property is trivially true for \( s_1^\perp = \bot_S \). Consider now the case where \( s_1^\perp = [\sigma, \bot_V], \sigma \neq \bot_V \). We now have two sub-cases: (a) \( s_2^\perp = [\tau, \bot_V], \tau \subseteq_V \sigma \) (b) \( s_2^\perp = [t_1, t_2], \bot_V \sqcup_V t_2, \sigma \subseteq_V t_1 \). The subcase (a) directly follows from the definition of \( \gamma_S \) applied to \( s_1^\perp \) and \( s_2^\perp \) and the fact that \( \tau \subseteq_V \sigma \). For subcase (b), we have:

\[
\begin{align*}
\gamma_S(s_1^\perp) &= \gamma_S([\sigma, \bot_V]) \subseteq (\text{ because } \sigma \subseteq_V t_1) \\
&= \gamma_S([t_1, \bot_V]) \subseteq (\text{ from subcase (a)}) \\
&= \gamma_S([t_1, t_2]) = \gamma_S(s_2^\perp).
\end{align*}
\]

The last case happens when both \( s_1^\perp \) and \( s_2^\perp \) have non-bottom elements. If \( s_1^\perp = [\sigma_1, t_1] \) and \( s_2^\perp = [\sigma_2, t_2] \) then \( s_1^\perp \subseteq_S s_2^\perp \) implies \( \sigma_1 \subseteq_V \sigma_2 \) and \( t_1 \subseteq_V t_2 \). From these facts and the definition of \( \gamma_V \), the property directly follows.

(Galois) We must show that \( \forall y \in \mathcal{S}, y \subseteq \gamma_S(\alpha_S(y)) \). Besides the trivial \( y = \emptyset \) and \( y = \{\lambda\} \), consider these two cases: (a) \( y \) contains stacks with length at most 1; (b) \( y \) contains at least one stack whose length is greater than 1. In case (a), \( \alpha_V(y) = [\sigma, \bot_V] \) where \( \sigma = \bigcup_{(k, \tau) \in y} \tau \). Since \( \gamma_V([\sigma, \bot_V]) \) contains the empty stack plus all the stacks with length one whose top element has a security level lower or equal to \( \sigma \), then it must contain all the elements of \( y \). In case (b), \( \alpha_V(y) \) will be in the form \( [\sigma_1, \sigma_2] \), where \( \sigma_1 \) and \( \sigma_2 \) are the least upper bounds of the security levels of the top and non-top elements of stacks in \( y \), respectively. Now the concretization of \( \alpha_V(y) \) will contain all the concrete stacks (also those whose length is less than or equal to 1) whose elements have security levels lower than \( \sigma_1 \) (top elements) and \( \sigma_2 \) (other elements). This infinite set contains all the elements of \( y \).

(Insertion) To prove the insertion property, it is necessary to show that \( \forall s^\perp = S^\perp, \alpha_S(\gamma_S(s^\perp)) = s^\perp \). Again, we proceed by cases. Excluding the trivial \( s^\perp = \bot_S \) and \( s^\perp = [\bot_V, \bot_V] \) we can consider: (a) \( s^\perp = [\sigma, \bot_V] \); (b) \( s^\perp = [\sigma_1, \sigma_2], \bot_V \sqcup_V \sigma_2 \). In case (a) \( \gamma_S(s^\perp) = \{\lambda\} \cup \{(k, t)| \tau \subseteq_L \sigma\} \), whose abstraction is clearly again \( s^\perp \). In case (b), the expression for \( \gamma_S(s^\perp) \) is a bit more complicated by the presence of the 1-element stacks; however the property clearly holds. \( \square \)
A.4. Proof of Theorem 5.7

The following proposition states that the abstraction of a set of concrete states is the least upper bound of the abstractions of the single elements of the set.

**Proposition A.1.** \( \forall Q \in \wp(Q), \)
\[
\alpha_Q(Q) = \bigsqcup_{q \in Q} \alpha_Q(q)
\]

**Proof.** By definition of \( \alpha_Q. \) \( \Box \)

**Theorem 5.7.** \( \alpha_Q \) and \( \gamma_Q \) form a Galois insertion.

**Proof.**

(\( \omega \)-Monotonicity) We want to prove that, for all \( Q, Q' \in \wp(Q), Q \subseteq Q' \Rightarrow \alpha_Q(Q) \subseteq \alpha_Q(Q'). \) From the definition of \( \alpha_Q, \) it is straightforward to prove \( \text{dom}(\alpha_Q(Q)) \subseteq \text{dom}(\alpha_Q(Q')). \) It remains to prove that, given \( t \in B \) with \( \alpha_Q(Q)(t) = \{\sigma, \mu, s, c\}, \) \( \alpha_Q(Q')(t) = \{\sigma', \mu', s', c'\}, \) it is true that: (1) \( \sigma \sqsubseteq \sigma', \) (2) \( \mu \sqsubseteq \mu', \) (3) \( s \sqsubseteq s', \) and (4) \( c \sqsubseteq c'. \) The first inequality holds by definition of \( \alpha_Q \) and the set inclusion between \( Q \) and \( Q'. \) The other three are proven by the fact \( Q \subseteq Q' \Rightarrow \{q(t) \mid q(t) \in Q\} \subseteq \{q(t) \mid q(t) \in Q'\}. \)

(\( \gamma \)-Monotonicity) To prove this property, we must show that \( \forall q_1, q_2 \in Q, q_1 \sqsubseteq q_2, \gamma_Q(q_1) \subseteq \gamma_Q(q_2). \) Firstly, we note that \( q_1 \sqsubseteq q_2 \) implies \( \text{dom}(\gamma_Q(q_1)) \subseteq \text{dom}(\gamma_Q(q_2)). \) Other consequences of the hypothesis are that, if \( q^t_1(t) = (\sigma_1, \mu_1, s_1, c_1^t) \) and \( q^t_2(t) = (\sigma_2, \mu_2, s_2, c_2^t), \) then \( \sigma_1 \sqsubseteq \sigma_2, \mu_1 \sqsubseteq \mu_2, s_1 \sqsubseteq s_2 \) and \( c_1^t \sqsubseteq c_2^t. \) By \( \gamma \)-monotonicity for \( M, S, \) and \( C, \) we have that \( \gamma_M(\mu_1^t) \subseteq \gamma_M(\mu_2^t), \gamma_S(s_1^t) \subseteq \gamma_S(s_2^t) \) and \( \gamma_C(c_1^t) \subseteq \gamma_C(c_2^t) \). All these facts suffice to state that, by construction of \( \gamma_Q, \gamma_Q(q_1) \subseteq \gamma_Q(q_2). \)

(Galois) Let \( Q \in \wp(Q) \) and \( q \in Q, q = (t, \rho, \mu, s, c). \) We will show that \( q \in \gamma_Q(\alpha_Q(Q)) \) to prove that \( Q \subseteq \gamma_Q(\alpha_Q(Q)). \)

\[
\begin{align*}
q \in Q & \Rightarrow \{q\} \subseteq Q & \Rightarrow \text{by monotonicity of } \alpha_Q \\
\Rightarrow \alpha_Q(q) \subseteq Q & \Rightarrow \gamma_Q(\alpha_Q(q)) \subseteq \gamma_Q(\alpha_Q(Q)). & \Rightarrow \text{by monotonicity of } \gamma_Q
\end{align*}
\]

Thus, it suffices to prove that \( q \in \gamma_Q(\alpha_Q(\{q\})). \) We have that \( \alpha_Q(\{q\}) = q^c, \) with \( \text{dom}(q^c) = \{t\} \) and \( \alpha_Q(\{q\})(t) = (\rho(t), \alpha_M(\mu), \alpha_S(s), \alpha_C(c)). \) Applying Propositions 5.2 and 5.6 we have that \( q \in \gamma_Q(\alpha_Q(\{q\})) = \{q'(t, \rho', \mu', s', c') \mid \forall t' \in B, \rho'(t) \sqsubseteq \rho(t), \mu'(t) \sqsubseteq \mu(t), s'(t) \sqsubseteq s(t) \} \) which clearly contains \( q. \)

**insertion** We have to prove that \( \forall q^2 \in Q, \alpha_Q(\gamma_Q(q^2)) = q^c. \) We will show that the property holds analyzing each field of a generic abstract state for the instruction \( t. \) For the environment we have:

\[
\begin{align*}
\alpha_Q(\gamma_Q(q^2)|(t).env) & = \bigsqcup_{\rho(t) \sqsubseteq \rho^c} \{t', \rho, \mu, s, c\} \in \gamma_Q(q^2) & = \text{by definition of } \alpha_Q \\
& = \bigsqcup_{\rho(t) \sqsubseteq \rho^c} q^c.env & = q^c.env
\end{align*}
\]

and for the memory:

\[
\begin{align*}
\alpha_Q(\gamma_Q(q^2)|(t).mem) & = \alpha_M(\{t, \rho, \mu, c, s\} \in \gamma_Q(q^2)) & = \text{by definition of } \alpha_Q \\
& = \alpha_M(\gamma_Q(q^2).mem) & = \text{by Proposition 5.2} \\
& = q^c.\text{mem}
\end{align*}
\]

The stack and channels parts are omitted. \( \Box \)
A.5. Proof of Theorem 6.5

We now prove the correctness of the abstract semantics, with respect to the concrete one. The following proposition states that each abstract rule correctly approximates its corresponding concrete version.

**Proposition A.2 (Arrow correspondence).**

\[
\forall q \in Q : q \xrightarrow{t} q' \Rightarrow \alpha_Q([q]) \xrightarrow{\overline{t}} q'' \text{ such that } \alpha_Q([q,q']) \subseteq Q q''
\]

**Proof.** Let us suppose \( q = \langle t, \rho, \mu, s, c \rangle, q' = \langle t', \rho', \mu', s', c' \rangle \). If \( t = t' \) the proof is trivial. Let \( t \neq t' \). From the definition of abstraction function we have that \( \text{dom}(\alpha_Q([q])) = \{ t \} \), \( \text{dom}(\alpha_Q([q,q'])) = \{ t, t' \} \). Moreover, \( \alpha_Q([q,q']) (t) = \alpha_Q([q]) \) since \( t \neq t' \). For each instruction \( t \not\in \{ t, t' \} \) it holds that, \( \alpha_Q([q,q']) (\overline{t}) = q'' (\overline{t}) \). Let \( \alpha([q,q']) (t) = \{ \sigma, \mu, s, c \} \), where \( \sigma = \rho (t) \). Now, since by the concrete semantics rules, \( \rho \subseteq_E \rho' \) and \( t \not\in \text{dom}(\alpha_Q([q])) \), then \( \alpha_Q([q,q']) (\overline{t'}) = \alpha_Q([q']) (\overline{t'}) \) What remains to prove is that \( \alpha_Q([q']) (\overline{t'}) \) is less than or equal to \( q'' (\overline{t'}) \). We proceed by cases that depend on the type of instruction with label \( t \).

- **(op)** Omitted.
- **(push)** This instruction only affects the status of the stack. Then, according to the rules, \( s' = \text{push}((k, \rho (t), \sigma), s) \), \( s'' = \text{push}^2(\rho (t), s') \) and we have to prove that \( \alpha_S(s') \subseteq_S s'' \). This directly follows from the Proposition 5.5(i), since \( s'' = \alpha_S(s) \) by hypothesis.
- **(pop)** Again, this instruction only affects the status of the stack: \( s' = \text{pop}(s), s'' = \text{push}^2(s') \) and the proof that \( \alpha_S(s') \subseteq_S s'' \) is obtained applying Proposition 5.5(ii).
- **(load)** Again, only the stack changes after the execution of this instruction. We have: \( s' = \text{push}((k, \rho (t) \cup \sigma), s) \) if \( \mu (x) = (k, \sigma') \) and \( s'' = \text{push}^2(\mu^2 (x) \cup \rho (t), s') \). On the other hand, since \( \mu^2 = \alpha_M(\mu) \), then \( \mu^2 (x) = \alpha_M(\mu (x)) = \alpha_M(k, \sigma') = \sigma' \). Therefore, by applying Proposition 5.5(i) we can prove \( \alpha_S(s') \subseteq_S s'' \).
- **(store)** This instruction changes both the stack and the memory. From the point of view of the stack, the instruction acts like a \( \text{pop} \), so we can reuse the proof for the case (pop). For the memory we have: \( \mu' = \mu - \{ x \leftarrow (k, \sigma' \cup \rho (t)) \} \) if \( \text{top}(s) = (k, \sigma') \) and \( \mu'^2 = \mu^2 - \{ x \leftarrow \text{top}(s^2) \cup \rho (t) \} \). Since \( s^2 = \alpha_S(s) \) by hypothesis, then \( \text{top}(s^2) \) is \( \sigma' \). To prove \( \alpha_M(\mu') \subseteq \mu'^2 \) we can observe that they may differ only in terms of the variable \( x \) for which it holds that \( \alpha_M(\mu' (x)) = \alpha_M(\mu (x)) = \alpha_M(k, \sigma' \cup \rho (t)) = (\sigma' \cup \rho (t)) = \mu^2 (x) \).
- **(send)** In this case, besides the stack (for which this instruction behaves like a \( \text{pop} \)), the status of the channels changes as well. Suppose that we send the value to channel \( a \), with \( c(a) = (k, \delta, \sigma') \) and \( \text{top}(c) = (k, \sigma') \). We have: \( c' (a) = (k, \delta, \tau \cup_C \sigma') \) and therefore \( \alpha_C(c (a)) = \tau \cup_C \sigma' \cup_C \rho (t) \). Now, by abstracting we obtain \( c'' (a) = \tau \) and, if we apply the abstract rule \( c'' (a) \) is less than or equal to \( \alpha_C(c (a)) \). Since \( \sigma = \rho (t) \) by hypothesis and the other channels are unchanged, this case is proved.
- **(receive)** For the stack, this subproof is similar to the (load) case. Suppose that we receive a value from the channel \( a \) and \( c(a) = (k, \delta, \sigma') \). We have: \( s' = \text{push}((k, \rho (t) \cup \sigma), s) \) and \( s'' = \text{push}^2(c^2 (a) \cup \rho (t), s') \). On the other hand, since \( c^2 (a) = \alpha_C(c) \), then \( c^2 (a) = \sigma' \). Therefore, if we apply Proposition 5.5(i) we can prove \( \alpha_S(s') \subseteq_S s'' \). We can proceed similarly for the change in the channel status.
- **(goto)** Omitted.
- **(if)** Since the execution of an \( i \notin \) instruction pops the tested value, for the stack we can reuse the proof for the case (pop). If the top of the stack \( s \) has annotation \( \tau' \) then the abstract stack \( s'' = \alpha_S(s) \) is in the form \( [\tau' \cdot v^2_1 \cdots v^2_{n-1}] \), and therefore the environment is upgraded by \( \xrightarrow{\overline{t}} \) and \( \xrightarrow{\overline{t}} \) in the same way.
- **(halt)** Omitted.

**Lemma A.3.** The application of lift preserves abstraction. That is, \( \forall Q \in \wp(Q), \alpha_Q(\text{lift}(Q)) = \alpha_Q(Q) \)
Proof sketch. Since lift(Q) contains all the states in Q, plus some states for which the security levels are lifted to the least upper bound of states in Q, these latter states do not contribute to the least upper bound in the application of αQ. □

Proposition A.4 (Local correctness). next is a safe approximation of next :

\[ \forall Q \in \wp(Q) : \text{next}(Q) \subseteq \gamma(\text{next}^\oplus(\alpha(Q))) \]

Proof. It suffices to prove that:

\[ \alpha(\text{next}(Q)) \subseteq \text{next}^\oplus(\alpha(Q)) \] (*)

Indeed, applying \( \gamma \)-monotonicity to (*), we can conclude:

\[ \text{next}(Q) \subseteq \gamma(\alpha(\text{next}(Q))) \subseteq \gamma(\text{next}^\oplus(\alpha(Q))) \]

where the first subset operation is given by the Galois property (Proposition 5.7).

Statement (*) can be derived directly from Proposition A.2 and from the definitions of next and next^\oplus operators. Using Proposition A.1 and the definition of next^\oplus, we can rewrite the right-hand member of (*) as:

\[ \text{next}^\oplus(\alpha(Q)) = \text{next}^\oplus\left( \bigcup_{q'' \in Q} \alpha(\{q''\}) \right) = \bigcup_{q''} q'' \]

On the other hand, we can use the definition of next to obtain:

\[
\begin{align*}
\alpha_Q(\text{next}(Q)) &= \alpha_Q(\text{lift}(Q \cup \{q' \mid \exists q \in Q : q \xrightarrow{l} q'\})) = \text{(by Lemma A.3)} \\
&= \alpha_Q(Q \cup \{q' \mid \exists q \in Q : q \xrightarrow{l} q'\}) = \\
&= \bigcup_{q \rightarrow q', q' \in Q} \alpha_Q(\{q, q'\})
\end{align*}
\]

From these two results we can rewrite (*) as:

\[ \bigcup_{q \rightarrow q', q' \in Q} \alpha_Q(\{q, q'\}) \subseteq \bigcup_{q''} q'' \]

Since Proposition A.2 holds, each state contributing to the left-hand lub is certainly less than or equal to a state contributing to the right-hand lub. The lattice properties ensure that, given \( a, b, c, d \) in a lattice \( (A, \sqsubseteq) \), with \( a \sqsubseteq b \land c \sqsubseteq d \), then \( (a \cup c) \sqsubseteq (b \cup d) \). By applying this property, we can conclude (**) and therefore the thesis.

□

Now, recalling Theorem 6.5:

Theorem 6.5 (Global correctness). \( \alpha_Q(\text{sem}) \subseteq \text{sem}^\oplus \).

Proof. Using the Local Correctness and proving by induction that for every initial configuration of the input channels \( i_0 \in \text{Names}_I \rightarrow \mathbb{Z}^* \), \( \forall j, \alpha_Q(\text{sem}_j(i_0)) \subseteq Q \text{ sem}^\oplus_j \). □
References


