Under consideration for publication in Math. Struct. in Comp. Science

Modelling Declassification Policies using Abstract Domain Completeness

Isabella Mastroeni[a] and Anindya Banerjee[b]

[a] *Università di Verona, Verona, Italy*
e-mail:isabella.mastroeni@univr.it
[b] *IMDEA Software Institute, Madrid, Spain*
e-mail: anindya.banerjee@imdea.org

Received 13 October 2010

This paper explores a three dimensional characterization of a declassification-based noninterference policy and its consequences. Two of the dimensions consist in specifying (a) the power of the attacker, that is, what public information an attacker can observe of a program, and (b) what secret information of a program needs to be protected. Both these dimensions are regulated by the third dimension, (c) the choice of program semantics, for example, trace semantics or denotational semantics, or, for instance, any semantics in Cousot's semantics hierarchy.

To check whether a program satisfies a noninterference policy one can compute an abstract domain that over-approximates the information released by the policy and can subsequently check whether program execution may release more information than what is permitted by the policy. Counterexamples to a policy can be generated by using a variant of the Paige-Tarjan algorithm for partition refinement. Given the counterexamples the policy can be refined so that the least amount of confidential information necessary for making the program secure is declassified.

1. Introduction

The secure information flow problem is concerned with protecting data confidentiality by checking that secrets are not leaked during program execution. In the simplest setting, data channels such as program variables, object fields, etc. are annotated with security labels, `H` (high/private/secret) and `L` (low/public/observable) where the labels denote levels in a two point lattice, $\mathtt{L} \leq \mathtt{H}$. Protection of data confidentiality amounts to ensuring that `L` outputs are not influenced by `H` inputs (Cohen 1977).

The labels serve to regulate information flow based on the *mandatory access control* assumption, namely, that a principal can access a `H` channel only if it is authorized to access `H` channels. The assumption leads to the classic "no reads up" and "no writes down" information flow control polices of (Bell and LaPadula 1973): a principal at level `L` (`H`) is authorized to read only from channels at level `L` (`H` and `L`) and a principal at level `H` is not authorized to write to a channel at level `L`. A more formal description of the Bell and La Padula policies is *noninterference* (NI) (Goguen and Meseguer 1984):

for any two runs of a program, `L` indistinguishable input states yield `L` indistinguishable output states, where two program states are said to be `L` indistinguishable iff they agree on the values of the `L` variables, but not necessarily on the values of `H` variables. Ergo, `L` outputs are not influenced by `H` inputs.

NI is enforced by an information flow analysis. Enforcement involves checking that information about initial `H` inputs of a program do not flow to `L` output variables either directly, by way of data flow, or implicitly, by way of control flow. One can consider leaks of initial `H` inputs via other covert channels like power consumption or memory exhaustion — but such modes of information transmission are not considered in this paper. A variety of information flow analyses have been formulated using technologies such as data flow analysis, security type systems, program logics, etc. (See the survey by (Sabelfeld and Myers 2003) and references therein).

The above discussion implicitly assumes that given a program which manipulates mixed data — some secret and some public — there is an attacker (a principal at level `L`) trying to obtain information about secret program inputs. But how can the attacker discover this information absent its ability to directly lookup values of secret input variables?

1.1. *First contribution*

We address the above question by suggesting that a NI policy is characterized by three dimensions: (a) the observation policy, that is, what the attacker can observe/discover (also termed the "attacker model") (Sec. 3.2) and (b) what must be protected — or dually, declassified — which we term the "protection policy" or dually, the "declassification policy" (Sec. 3.3), (c) choice of the concrete semantics of a program (Sec. 3.1).

For (c), there is a hierarchy of semantics from which to choose (Cousot 2002): two example elements in the hierarchy are trace semantics and denotational semantics with the latter being an abstraction of the former. But a concrete semantics represents an infinite mathematical object and by Rice's theorem all non-trivial questions about the concrete semantics are undecidable. Thus, for (a) it is clear that the attacker can only observe/discover, by way of program analyses, *approximated, abstract properties* of secret inputs and public outputs rather than their *exact, concrete* values. For example, given a program that manipulates numeric values, the attacker might discover — using an Octagon abstract domain (Miné 2006)— that a particular secret input, say h, lies in a range $5 \leq h \leq 8$. The choice of semantics also delimits *where* the attacker might be able to make the observations, e.g., only at input-output (for denotational semantics) or all intermediate program points (for trace semantics).

Finally, (b) specifies the protection policy, namely, what property of the initial secret input must be protected. Often this is stated dually, by making explicit what property of the initial secret may be released or *declassified.* For example, the policy may say, as in classic NI, that nothing may be declassified. On the other hand, an organization may enroll in a statistical survey that would require declassifying the average salary of its employees. Here again, the choice of semantics plays a crucial role: for example, with a denotational semantics we only care that the declassification policy be honored at the output, rather than at intermediate program points (in contrast to a trace semantics).

1.2. *Second contribution*

Our second contribution (Sects. 4,5) is to show that given a fixed program semantics, the declassification policy-dimension of NI can be expressed as a *completeness problem in abstract interpretation*: the program points where completeness *fails* are the ones where some secret information *is leaked*, thus breaking the policy. Hence, we can check if a program satisfies a declassification policy by checking whether its semantics is complete w.r.t. the policy.

Moreover, we show that when a program does not satisfy a declassification policy (i.e., when completeness fails), (i) counterexamples that expose the failure can be generated (Sec. 7); (ii) there is an algorithm that generates the best refinement of the given policy such that the program respects the refined policy (Sec. 7.1).

1.3. *Third contribution*

We provide an algorithmic approach to refinement of declassification policies and to counterexample generation.

To achieve this we connect declassification and completeness (Sec. 5) on the one hand, to completeness and stability, which themselves have been related via the Paige-Tarjan algorithm (Paige and Tarjan 1987; Ranzato and Tapparo 2005; Mastroeni 2008). The upshot is that we are able to show, in Proposition 7.1, that the absence of unstable elements in the program input domain (in the Paige-Tarjan sense) corresponds to the absence of information leaks in declassified noninterference (Sec. 8). This relation is shown also by means of examples, where we use the Paige-Tarjan algorithm for revealing information leaks and for refining the corresponding declassification policy.

Finally we create a bridge between declassified noninterference and abstract model checking: the absence of spurious counterexamples in abstract model checking can be understood as the absence of information leaks in declassified noninterference.

1.4. *Paper road map*

In Sec. 3 we discuss how we can describe a noninterference policy in terms of three different dimensions: semantic, observation and protection policies. In particular we focus on the semantic policy which determines the ground where we can fix what an attacker can observe (observation policy) and what we aim to protect (protection policy), see Fig. 2. At this point we introduce a technique based on two main ingredients: completeness in abstract interpretation and weakest precondition semantics, whose relation with noninterference is explained in Sec. 4. Hence in Sec. 5 we explain how we can check declassified noninterference policies by using a weakest precondition semantics-based technique whose theoretical foundations are based on abstract interpretation completeness. In particular we introduce this technique on policies defined in terms of I/O semantics, i.e., where the attacker can only observe the public input and the public output. In order to show that this technique can be used in the more general formalization of noninterference, the one proposed in Sec. 3 in terms of three dimensions, we show, in Sec. 6 how we can easily extend the technique to noninterference policies defined in terms of the trace semantics, where the attacker can also observe intermediate public states. Finally, we explain how we can use our technique for characterizing counterexamples to a given declassified noninterference policy (Sec. 7) and how we can refine a given declassified policy in order to characterize the most restrictive declassified policy satisfied by the program (Sec. 8). The interesting aspect or our technique is that these last characterizations can be used in the more general formalization of noninterference since they are independent from the semantics, in fact the semantics characterizes *what* the attacker can use for attacking a program (observed public information), but not how it uses this information, which depends only on what it has deduced from its observations.

2. Background

This section considers the semantics of Winskel's simple imperative language IMP (Winskel 1993), introduces relevant terminology and reviews completeness of abstract interpretations.

2.1. *The imperative language*

Syntax of IMP. The syntactic categories associated with IMP are: Numerical values $\mathbb{V}$; Truth values $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$; Variables *Var*; Arithmetic expression *Aexp*; Boolean expression *Bexp*; Commands *Com*.

We assume that the syntactic structure of numbers is given. We will use the following convention: m, n range over values $\mathbb{V}$; x, y range over variables *Var*; a ranges over arithmetic expression *Aexp*; b ranges over boolean expression *Bexp*; c ranges over commands *Com*. We describe the arithmetic and boolean expressions in *Aexp* and *Bexp* as follows:

$$\begin{array}{lll} a & ::= & n \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \cdot a_1 \\ b & ::= & \mathit{true} \mid \mathit{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \end{array}$$

Finally, for commands we have the following syntax:

$$c ::= \textbf{skip} \mid x := a \mid c_0; c_1 \mid \textbf{while } b \textbf{ do } c \textbf{ endw} \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1$$

A complete program is a command. In the sequel, we will consider a slight extension of this language with explicit syntax for declassification as in Jif (Myers et al. 2001) or in the work on delimited release (Sabelfeld and Myers 2004). Such syntax often takes the form $l := \mathit{declassify}(h)$, where l is a public variable and h is a secret variable.

Semantics. $\mathbb{V}$ is structured as a flat domain with additional bottom element $\bot$, denoting the value of uninitialized variables. We denote by $\mathit{Vars}(P)$ the set of variables of the program $P \in$ IMP. The standard big-step semantics of IMP is given by Fig. 1. In the figure, the notation $\langle c, s\rangle \rightarrow s'$ says that command c transforms state s to state s', where a state associates a variable to a value. State s' is termed the successor of state s. The semantics naturally induces a transition relation, $\rightarrow$, on a set of states, Σ, specifying the relation between a state and its possible successors. We say that $\langle \Sigma, \rightarrow \rangle$ is a transition system. States are represented as tuples of values indexed by variables. For example, if $|\mathit{Vars}(P)| = n$, then Σ is a set of n-tuples of values, i.e., $\Sigma = \mathbb{V}^n$. We abuse notation by denoting with $\bot$ the state where all the variables are undefined.

$$\langle \mathbf{skip}, s\rangle \rightarrow s \qquad \frac{\langle e, s\rangle \rightarrow n \in \mathbb{V}_x}{\langle x := e, s\rangle \rightarrow s[x \mapsto n]} \qquad \frac{\langle c_0, s\rangle \rightarrow s',\ \langle c_1, s'\rangle \rightarrow s''}{\langle c_0; c_1, s\rangle \rightarrow s''}$$

$$\frac{\langle b, s\rangle \rightarrow \mathit{true},\ \langle c_0, s\rangle \rightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, s\rangle \rightarrow s'} \qquad \frac{\langle b, s\rangle \rightarrow \mathit{false},\ \langle c_1, s\rangle \rightarrow s'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, s\rangle \rightarrow s'}$$

$$\frac{\langle \mathbf{if}\ b\ \mathbf{then}\ (c; \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw})\ \mathbf{else\ skip}, s\rangle \rightarrow s'}{\langle \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{endw}, s\rangle \rightarrow s'}$$

Fig. 1. Big-step semantics of Imp

Traces, maximal trace semantics. In the following, Σ^+ and $\Sigma^\omega \overset{\text{def}}{=} \mathbb{N} \longrightarrow \Sigma$ denote respectively the non-empty set of finite and infinite sequences of symbols in Σ. Given a sequence $\sigma \in \Sigma^\infty \overset{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$, its length is denoted by $|\sigma| \in \mathbb{N} \cup \{\omega\}$ and its i-th element is denoted by σ_i. A non-empty finite (infinite) *trace* $\sigma \in \Sigma^\infty$ is a finite (infinite) sequence of program states where two consecutive elements are in the transition relation $\rightarrow$, i.e., for all $i < |\sigma|$: $\sigma_i \rightarrow \sigma_{i+1}$. If $\sigma \in \Sigma^+$, then $\sigma_\dashv$ and $\sigma_\vdash$ denote respectively the final and initial state of σ. The *maximal trace semantics* (Cousot 2002) of a transition system associated with a program P is $\langle\!| P |\!\rangle \overset{\text{def}}{=} \langle\!| P |\!\rangle^+ \cup \langle\!| P |\!\rangle^\omega$, where $\Sigma_\dashv \subseteq \Sigma$ is a set of final/blocking states and $\Sigma_\vdash$ denotes the set of initial states for P. Then $\langle\!| P |\!\rangle^\omega = \{\sigma \in \Sigma^\omega \mid \forall i \in \mathbb{N} \,.\, \sigma_i \rightarrow \sigma_{i+1}\}$, $\langle\!| P |\!\rangle^+ = \{\sigma \in \Sigma^+ \mid \sigma_\dashv \in \Sigma_\dashv,\ \forall i \in [1, |\sigma|).\, \sigma_{i-1} \rightarrow \sigma_i\}$.
The denotational semantics is obtained by abstracting trace semantics to the input and output states only, i.e.,

$$[\![P]\!] = \lambda \sigma_\vdash . \begin{cases} \sigma_\dashv & \text{if } \sigma \in \langle\!| P |\!\rangle^+ \\ \bot & \text{if } \sigma \in \langle\!| P |\!\rangle^\omega \end{cases}$$

In this case, it is assumed that the output observation of an infinite computation is the state where all the variables are undefined (Cousot 2002).

In the following, we will also use the weakest precondition semantics (wlp for short) (Dijkstra 1975), isomorphic to the denotational one (Cousot 2002). Weakest precondition semantics, denoted *Wlp*, associates with each terminating program P and set of valid post-conditions Φ, the weakest set of pre-condition Φ' leading to Φ after the execution of P. In general the post and the pre conditions are set of states, hence $\mathit{Wlp}(P, \Phi) = \Phi'$ means that Φ' is the greatest set of states leading to the states Φ, by executing P.

2.2. *Review: Completeness of abstract interpretation*

Abstract interpretation is typically formulated using Galois connections (GC) (Cousot and Cousot 1977), but an equivalent framework (Cousot and Cousot 1979) which we use in this paper, uses *upper closure operators*. An *upper closure operator (uco)* $\rho : C \to C$ on a poset C is monotone, idempotent, and extensive, i.e., $\forall x \in C.\ x \leq_C \rho(x)$. The set of all upper closure operators on C is denoted by $uco(C)$.

In Example 2.1, $\mathcal{H} : \wp(\Sigma) \to \wp(\Sigma)$ defined as $\mathcal{H}(X) = \mathbb{V}^{\mathtt{H}} \times X^{\mathtt{L}}$, is an upper closure operator on $\wp(\Sigma)$, because $\mathcal{H}$ is monotone, idempotent and extensive. We often call a closure operator an *abstract domain*. In particular, $\mathcal{H}$ is called the *output (i.e., observed) abstract domain*, that ignores secret information.

Completeness of abstract interpretation based static analysis has its origins in Cousot's work, e.g., (Cousot and Cousot 1977; Cousot and Cousot 1979), and means that the analysis is as expressive as possible. The following example is taken from Schmidt's excellent survey (Schmidt 2006) on completeness. To validate the Hoare triple, $\{?\}\ y := -y; x := y+1\ \{\mathit{isPositive}(x)\}$, a *sound* analysis may compute the precondition $\mathit{isNegative}(y)$. But if able to express properties like *isNonNegative* and *isNonPositive*, a *complete* analysis will calculate the *weakest* precondition property $\mathit{isNonPositive}(y)$.

An abstract domain is complete for a concrete function, f, if the "abstract state transition function precisely mimics the concrete state-transition function modulo the GC between concrete and abstract domains" (Schmidt 2006). There exist two notions of completeness – *backward* ($\mathcal{B}$) and *forward* ($\mathcal{F}$) – according as whether the concrete and the abstract computations are compared in the abstract domain or in the concrete domain (Giacobazzi and Quintarelli 2001). Formally, let C be a complete lattice and f be the concrete state transition function, $f : C \to C$. Abstract domain ρ is a sound abstraction for f provided $\rho \circ f \circ \rho \sqsupseteq \rho \circ f$. For instance, in Example 2.1, $\mathcal{H}(\llbracket P \rrbracket(\mathcal{H}(X))) \supseteq \mathcal{H}(\llbracket P \rrbracket(X))$, so $\mathcal{H}$ is a sound abstraction for $\llbracket P \rrbracket$. Completeness is obtained by demanding equality: ρ is a $\mathcal{B}$ (resp. $\mathcal{F}$)-complete abstraction for f iff $\rho \circ f = \rho \circ f \circ \rho$ (resp. $f \circ \rho = \rho \circ f \circ \rho$). Completeness can be generalized to pairs (ρ, η) of abstract domains: $\mathcal{B}$-completeness holds for (ρ, η) when $\rho \circ f \circ \eta = \rho \circ f$; $\mathcal{F}$-completeness holds for (ρ, η) when $\rho \circ f \circ \eta = f \circ \eta$ (see (Giacobazzi et al. 2000) for details). Algorithms for completing abstract domains exist – see (Giacobazzi et al. 2000; Giacobazzi and Quintarelli 2001; Mastroeni 2008) and Schmidt's survey (Schmidt 2006) for details. Basically, $\mathcal{F}$-completeness is obtained by adding all the direct images of f to the output abstract domain; $\mathcal{B}$-completeness is obtained by adding all the maximal of the inverse images of the function to the input domain.

2.3. *On NI as a completeness problem*

By starting from Joshi and Leino's characterization of classic NI, (Giacobazzi and Mastroeni 2005) noted that classic NI is a completeness problem in abstract interpretation. (Joshi and Leino 2000) use a weakest precondition semantics of imperative programs to arrive at an equational definition of NI: a program P containing `H` and `L` variables (ranged over by h and l respectively) is secure iff

$$HH; P; HH = P; HH$$

where HH is an assignment of an arbitrary value to h. "The postfix occurrences of HH on each side mean that we are only interested in the final value of l and the prefix HH on the left-hand-side means that the two programs are equal if the final value of l does not depend on the initial value of h" (Sabelfeld and Sands 2001).

An abstract interpretation is (backwards) complete for a function, f, if the result obtained when f is applied to any concrete input, x, and the result obtained when f is applied to an abstraction of the concrete input, x, both abstract to the same value. Thus, the essence of completeness is this: an observer who can see only the final abstraction cannot distinguish whether the concrete input value was x or any other concrete value x' with the same abstract value as that of x. The completeness connection is implicit in Joshi and Leino's definition of secure information flow and the implicit abstraction in their definition is: "each $\mathtt{H}$ value is associated with $\top$, that is, the set of all possible $\mathtt{H}$ values". We now proceed to explain these ideas by way of an example.

Let $\mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}$ be the sets of possible $\mathtt{H}$ and $\mathtt{L}$ values. In particular, if $n = |\{\, x \in \mathit{Vars}(P) \mid x \text{ is } \mathtt{H} \,\}|$, then $\mathbb{V}^{\mathtt{H}} \overset{\text{def}}{=} \mathbb{V}^{n}$, analogously for $\mathtt{L}$ variables. The set of program states is $\Sigma = \mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}}$. Σ is implicitly indexed by the $\mathtt{H}$ variables followed by the $\mathtt{L}$ variables. For any $X \subseteq \Sigma$, $X^{\mathtt{H}}$ (resp. $X^{\mathtt{L}}$) is the projection of the $\mathtt{H}$ (resp. $\mathtt{L}$) variables. $\mathtt{L}$ indistinguishability of states $s_1, s_2 \in \Sigma$, written $s_1 =_{\mathtt{L}} s_2$, denotes that s_1, s_2 agree when indexed by $\mathtt{L}$ variables.

We start with Joshi and Leino's semantic definition of security (Joshi and Leino 2000), $HH; P; HH = P; HH$. Because HH is an arbitrary assignment to h, its semantics can be modelled as an *abstraction function*, $\mathcal{H}$, on sets of concrete program states, Σ; that is, $\mathcal{H} : \wp(\Sigma) \to \wp(\Sigma)$, where $\wp(\Sigma)$ is ordered by subset inclusion, $\subseteq$. For each possible value of an $\mathtt{L}$ variable, $\mathcal{H}$ associates *all* possible values of the $\mathtt{H}$ variables in P. Thus $\mathcal{H}(X) = \mathbb{V}^{\mathtt{H}} \times X^{\mathtt{L}}$, where $\mathbb{V}^{\mathtt{H}} = \top$, the top element of $\wp(\mathbb{V}^{\mathtt{H}})$. Now the Joshi-Leino definition can be rewritten (Giacobazzi and Mastroeni 2005) in the following way, where $[\![P]\!]$ is the concrete, denotational semantics of P.

$$\mathcal{H} \circ [\![P]\!] \circ \mathcal{H} = \mathcal{H} \circ [\![P]\!] \tag{1}$$

This is exactly the definition of backwards completeness in abstract interpretation (Cousot and Cousot 1979; Giacobazzi et al. 2000). Next example shows how we can interpret the completeness equation when noninterference does not hold.

Example 2.1. Let $h_1, h_2 \in \{0,1\}$ and let $l \in \{0,1\}$. Then $\mathbb{V}^{\mathtt{H}} = \{0,1\} \times \{0,1\}$, $\mathbb{V}^{\mathtt{L}} = \{0,1\}$. Consider any $X \subseteq \Sigma$; for example, let $X = \{\langle 0,0,1\rangle\}$, that is, X denotes the state where $h_1 = 0$, $h_2 = 0$, $l = 1$. Then $\mathcal{H}(X) = \mathbb{V}^{\mathtt{H}} \times \{1\}$. Let P be the obviously insecure program, $l := h_1$, so that, $[\![P]\!](X) = \{\langle 0,0,0\rangle\}$ and $\mathcal{H}([\![P]\!](X)) = \mathbb{V}^{\mathtt{H}} \times \{0\}$. On the other hand, $[\![P]\!](\mathcal{H}(X)) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \langle 1,0,1\rangle, \langle 1,1,1\rangle\}$ so that we have $\mathcal{H}([\![P]\!](\mathcal{H}(X))) = \mathbb{V}^{\mathtt{H}} \times \{0,1\}$; hence $\mathcal{H}([\![P]\!](\mathcal{H}(X))) \supseteq \mathcal{H}([\![P]\!](X))$. Because $\mathcal{H}([\![P]\!](\mathcal{H}(X)))$ contains triples $\langle 1,0,1\rangle$ and $\langle 1,1,1\rangle$ that are not present in $\mathcal{H}([\![P]\!](X))$, the dependence of l on h_1 has been exposed. In other words $\mathcal{H}[\![P]\!]\mathcal{H}$ collects in output all the possible observations due to the variation of h_1, while $\mathcal{H}[\![P]\!]$ provides the observation for a particular value h_1. Hence, if these two sets are different it means that we have a dependency of the observable output on the value of h_1. Thus P violates classic NI: for any two distinct values, 0 and 1 of h_1 in $\mathcal{H}([\![P]\!](\mathcal{H}(X)))$, two *distinct* values, 0 and 1, of l may be associated.

In this paper, we consider more flexible abstractions than the one considered by Joshi and Leino and show that such abstractions naturally describe declassification policies that are concerned with *what* information is declassified (Sabelfeld and Sands 2007).

3. The three dimensions of noninterference

Recall from the introduction that we alluded to the following three dimensions of NI: (a) the semantic policy, (b) the observation policy and (c) the protection/declassification policy. These dimensions are pictorially represented in Fig. 2. In general, to describe any

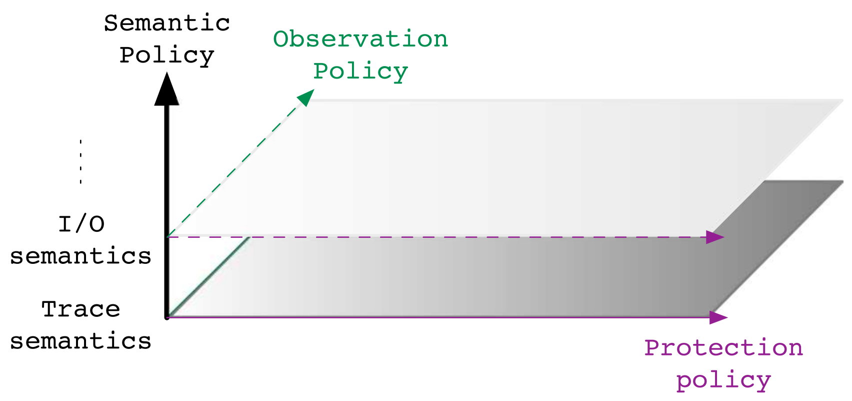


Fig. 2. The three dimensions of noninterference

NI policy for a program, we first fix its semantic policy. The semantic policy comprises of the *concrete semantics* of the program (the solid arrow in Fig. 2 shows Cousot's semantics hierarchy), the *set of observation points*, that is the program points where the attacker can observe data, and the *set of protection points*, that is the program points where information must be protected. Next we fix the program's observation policy and protection policy that say *what* information can be observed and *what* information needs to be protected at these program points. This is how we interpret the *what* dimension of declassification policies espoused by (Sabelfeld and Sands 2007).

As an example, consider classic NI, whose formal definition is given below (for a deterministic program P[1]):

$$\forall l \in \mathbb{V}^{\mathtt{L}}, h_1, h_2 \in \mathbb{V}^{\mathtt{H}}.\ [\![P]\!](h_1, l)^{\mathtt{L}} = [\![P]\!](h_2, l)^{\mathtt{L}}$$

Notice that the initial states (h_1, l) and (h_2, l) are L indistinguishable and the definition requires that the L projection of the final states be L indistinguishable.

For P above, the three dimensions of classic NI are as follows. For the semantic policy, the concrete semantics is P's denotational semantics. Both inputs and outputs constitute the observation points while inputs constitute the protection points because it is secret data at program inputs that need to be protected. The observation policy is the identity on the public input-output because the attacker only can observe L inputs and L outputs. Such an attacker is the most powerful attacker for the chosen concrete semantics because its knowledge (that is, the L projections of the initial and final states of the two runs of the program) is completely given by the concrete semantics . Finally, the protection policy is the identity on the *secret input*. This means that *all* secret inputs must be protected, or dually, *no* secret inputs must be declassified.

3.1. *The semantic policy*

In this section our goal is to provide a general description of a semantic policy that is parametric on any set of protection and observation points. For this purpose it is natural to move to a trace semantics. Our first step in the direction is to consider a function *post* as defined below.

Let $\to_P$ be the transition relation induced by the big-step semantics (Fig. 1) of a program P. Define

$$post_P \overset{\text{def}}{=} \left\{ \langle s, t\rangle \,\middle|\, s, t \in \Sigma,\ s \to_P t \right\}$$

From this definition of $post_P$ we can recover classic NI in the following way. First define the input-output relation below, which forms a function as P is deterministic[2]. Let $\to_P^i$ be the transitive composition of $\to_P$, i times. This implies, by construction, that $\to_P^i$ corresponds to the result in the i-th program point.

$$post_P^+ \overset{\text{def}}{=} \left\{ \langle s, t\rangle \,\middle|\, s \in \Sigma_{\vdash}, t \in \Sigma_{\dashv}, \exists i.\ s \to_P^i t \right\}$$

In other words, $post_P^+$ associates with each initial state s the final state t reachable from s. Then it is straightforward to note that an equivalent characterization of classic NI (that uses denotational semantics) is given by

$$\forall s_1, s_2 \in \Sigma_{\vdash}.\ s_1 =_{\mathtt{L}} s_2 \ \Rightarrow\ post_P^+(s_1) =_{\mathtt{L}} post_P^+(s_2)$$

where the set of initial states is $\Sigma_{\vdash}$. We will now use *post* to consider NI for trace semantics.

[1] If P is not deterministic the definition works anyway simply by interpreting $[\![P]\!](s)$ as the set of all the possible outputs starting from s.

[2] If P is not deterministic then the only thing to note is that we have to define the function *post* as a set of tuples $\langle s, T\rangle$, where T is the set of all the final states reachable from s. The observation is similar for all the subsequent definitions of *post* functions. Moreover, note that these definitions trivially hold even if the program is not terminating.

NI for trace semantics. A denotational semantics does not take into account the whole history of computation, and thus restricts the kind of protection/declassification policies one can model. In order to handle more precise policies, that take into account *where* (Sabelfeld and Sands 2007) information is released in addition to *what* information is released we must consider a more concrete semantics, such as trace semantics. First, we define classic NI on traces, using $\langle\!| P |\!\rangle$ to denote the trace semantics of P:

$$\forall l \in \mathbb{V}^{\mathtt{L}}, \forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}.\ \langle\!| P |\!\rangle(h_1, l)^{\mathtt{L}} = \langle\!| P |\!\rangle(h_2, l)^{\mathtt{L}}$$

This definition says that given two $\mathtt{L}$ indistinguishable input states, (h_1, l) and (h_2, l), the two executions of P must generate two — both finite or both infinite — sequences of states in which the corresponding states in each sequence are $\mathtt{L}$ indistinguishable. Equivalently, we can use a set of *post* relations: to wit, for $i \in \mathsf{Nats}$ we define the family of relations

$$post^i_P \overset{\text{def}}{=} \left\{ \ \langle s, t\rangle \,\middle|\ t \in \Sigma,\ s \in \Sigma_{\vdash},\ s \rightarrow^i_P t \ \right\}$$

(Because P is deterministic, each $post^i_P$ is indeed a function.) The following result is straightforward.

Proposition 3.1. NI on traces holds iff for each program point, i, of program P, we have

$$\forall s_1, s_2 \in \Sigma_{\vdash}. s_1 =_{\mathtt{L}} s_2 \ \Rightarrow\ post^i_P(s_1)^{\mathtt{L}} = post^i_P(s_2)^{\mathtt{L}}.$$

Proof. Trivially holds by noting that $\langle\!| P |\!\rangle(s_1)^{\mathtt{L}} = \langle\!| P |\!\rangle(s_2)^{\mathtt{L}}$ holds iff $post^i_P(s_1) =_{\mathtt{L}} post^i_P(s_2)$ holds for each i. □

The *post* characterization of trace noninterference precisely identifies the observation points as the outputs of the *post* relations, that is, any possible intermediate state of computation and the protection points are identified as the inputs of the *post* relations, that is, the initial states. We will now show how to generalize the semantic policy following this schema.

General semantic policies. In the previous paragraph, we show how for denotational and trace semantics, we can define a corresponding set of *post* relations fixing protection and observation points. In order to understand how we can generalize this definition, let us consider a graphical representation of the situations considered in Fig. 3.

(i) In the first picture, the semantic policy says that an attacker can observe the public inputs and outputs only, while we can protect only the secret inputs. This notion corresponds to classic NI.

(ii) In the second picture, the semantic policy says that an attacker can observe each intermediate state of computation, including the input and the output, while the protection point is the same as in part (i) — only secret inputs must be protected. Any possible leakage of secret data in intermediate states is not taken into account as only the protection of secret input is mandated by the policy. This notion corresponds to the noninterference policy introduced in robust declassification (Zdancewic and

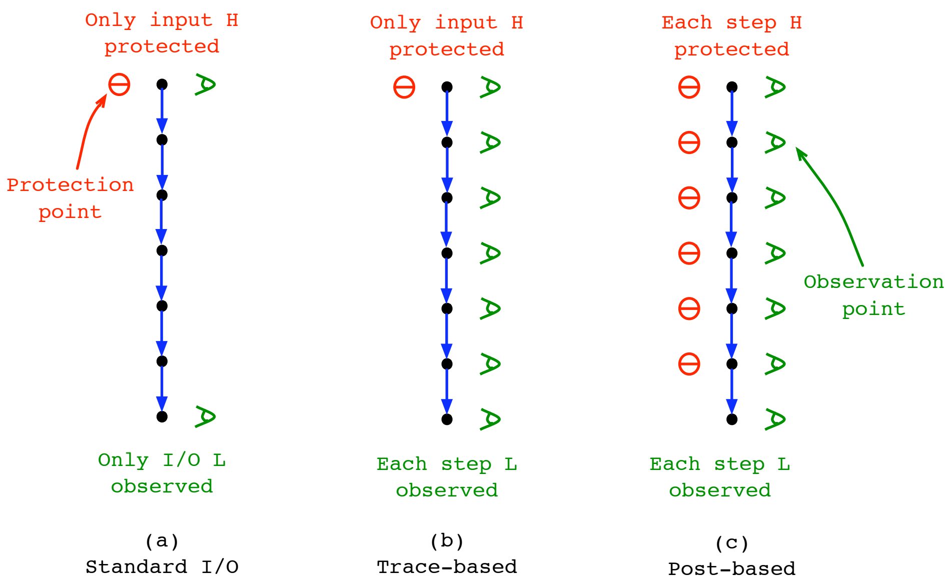


Fig. 3. Different notions of noninterference for trace semantics

Myers 2001), up to an abstraction of traces. This is the notion characterized above in terms of trace semantics.

(iii) In the last picture, we show another possible choice. In this case the semantic policy says that an attacker can observe each intermediate state of computation, while the protection points are all intermediate states of the computation. In order to check this notion of noninterference for a program we have to check noninterference separately for each statement of the program itself. It is worth noting that this corresponds exactly to $\forall s_1, s_2 \in \Sigma.\ s_1 =_{\mathtt{L}} s_2.\ post_P(s_1) =_{\mathtt{L}} post_P(s_2)$. Because P is deterministic, $post_P$ corresponds to an input-output function that can have any possible reachable state/program point as output (the attacker can observe any program point), and any possible state/program point as input (we want to protect secret data in any program point)[3]. Note that (iii) differs form (ii) only in interactive contexts, where there are intermediate private inputs to protect.

It is clear that between (i) and (ii) there are several notions of noninterference depending on the intermediate states of the computation the attacker can observe (*observation points* in Fig. 3). For example, gradual release (Askarov and Sabelfeld 2007a) considers as observation points only those program points corresponding to low events (i.e., assignment to low variables, declassification points and termination points). Likewise, between (ii) and (iii) there are several notions of noninterference, depending on points of the computation where we have to protect the secret part of the state (*protection points* in Fig. 3). In general, there are also several possibilities between (i) and (iii), in which

[3] This notion is also the one proposed for approximating abstract noninterference (Giacobazzi and Mastroeni 2004).

particular observation points as well as particular protection points are fixed. In order to model these intermediate possibilities consider the following relation, which is again a function for deterministic programs:

$$post_P^{i,j} \stackrel{\text{def}}{=} \left\{ \langle s,t\rangle \;\middle|\; s,t \in \Sigma,\ \exists s' \in \Sigma_\vdash .\ s' \rightarrow_P^i s \rightarrow_P^{j-i} t \right\}$$

In other words, i is the program point where the secret input can change, so we want to protect the secret input at program point i, while j is the program point where the attacker can observe the computation. Hence, consider a set $\mathtt{P}$ of protection points and the set $\mathtt{O}$ of observation points, both sets of indexes such that 0 corresponds to the initial program point. In the following, for each index $i \in \mathtt{P}$ we denote by p_i the corresponding protection point, while for $i \in \mathtt{O}$ we denote by o_i the corresponding observation point. Now, we can define a notion of NI parametric on these two sets of program points in the following way: Consider any i in $\mathtt{P}$ and any j in $\mathtt{O}$ with $i < j$. Further, consider any $s_1, s_2 \in \Sigma_\vdash$ such that $s_1 \rightarrow_P^i s_1'$ and $s_2 \rightarrow_P^i s_2'$. Then:

$$s_1' =_{\mathtt{L}} s_2' \;\Rightarrow\; post_P^{i,j}(s_1') =_{\mathtt{L}} post_P^{i,j}(s_2')$$

In this way we are able to characterize also the *where* dimension of declassification, which allows us to specify where we want to protect secret data, e.g., in the input or in any possible intermediate state, and where the attacker can observe, e.g., in the input-output or in any intermediate state, thus revealing the states in which information leakage happens. However, as underlined before, unless we consider interactive systems that can provide inputs in arbitrary states, the secret inputs of a program in its initial state are the only interesting secrets to protect. Thus we will restrict ourselves to initial secrets in the sequel. Hence, in the definition above $\mathtt{P}$ will always be $\{0\}$, namely we will consider as input states to protect only the initial ones. Hence, (ii) and (iii) collapse to the same notion.

Definition 3.2 [Trace-based NI] Given a set $\mathtt{O}$ of observation points, the notion of NI based on the semantic policy with $\mathtt{O}$ as observation points, and the input as protection point is defined as follows:

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_\vdash .\ s_1 =_{\mathtt{L}} s_2 \;\Rightarrow\; post_P^j(s_1) =_{\mathtt{L}} post_P^j(s_2)$$

Note that, this is a general notion, such that any other NI notion, described in Sec. 3 can be formulated as an instantiation of it, depending on $\mathtt{O}$. In particular, we obtain standard NI by fixing $\mathtt{O} = \left\{ p \;\middle|\; p \text{ is the final program point } \right\}$, we obtain the most concrete trace-based NI policy by fixing $\mathtt{O} = \left\{ p \;\middle|\; p \text{ is any program point } \right\}$. But we can also obtain intermediate notions depending on $\mathtt{O}$: we obtain gradual release by fixing $\mathtt{O} = \left\{ p \;\middle|\; p \text{ is a program point corresponding to a low event } \right\}$. We can observe that, without intermediate inputs, the last two possibilities provide the same notion, namely observing all the states does not provide the attacker more information than observing only the low event states. Nevertheless, the parametric notion allows to model situations where the environment can, for example, avoid some low event to be observed.

Example 3.3. Let us consider the program fragment P with the semantic policies represented in the following picture. The semantic policy represented in picture (a) is the

I/O one. In this case, for each pair of initial states s_1, s_2 such that $s_1^{\mathtt{L}} = s_2^{\mathtt{L}}$ we have to check $post^5(s_1)^{\mathtt{L}} = post^5(s_2)^{\mathtt{L}}$, and it is clear that this does not hold since, for instance, if $s_1 = \langle h_1, h_2 = 2, l_1, l_2\rangle$ and $s_2 = \langle h_1, h_2 = 3, l_1, l_2\rangle$ then we have different l_2 values in output: respectively $l_2 = 0$ and $l_2 = 1$.

```
$p_0 \rightarrow$                     $\leftarrow o_0$
        $h_1 := h_2;$
        $h_2 := h_2$ mod $2;$
        $l_1 := h_2$
        $l_2 := h_1$
        $l_2 := l_1$
                                 $\leftarrow o_5$
        $(a)$
```

```
$p_0 \rightarrow$                     $\leftarrow o_0$
        $h_1 := h_2;$
        $h_2 := h_2$ mod $2;$
        $l_1 := h_2$
                                 $\leftarrow o_3$
        $l_2 := h_1$
                                 $\leftarrow o_4$
        $l_2 := l_1$
                                 $\leftarrow o_5$
        $(b)$
```

On the other hand, the policy in picture (b) considers $\mathtt{O} = \{3, 4, 5\}$ (the low events). In this case, for each pair of initial states s_1, s_2 such that $s_1^{\mathtt{L}} = s_2^{\mathtt{L}}$ we have to check $post^5(s_1)^{\mathtt{L}} = post^5(s_2)^{\mathtt{L}}$, but also $post^4(s_1)^{\mathtt{L}} = post^4(s_2)^{\mathtt{L}}$ and $post^3(s_1)^{\mathtt{L}} = post^3(s_2)^{\mathtt{L}}$, and all these tests fails since in all the corresponding program points there is a leakage of private information.

3.2. *The observation policy*

Suppose the observation points fixed by the chosen semantics are input and output. Then the observation policy might require that the attacker observes a particular *property*, ρ, of the public output — e.g., parity — and a particular property, η, of the public input — e.g., signs. (Technically, η, ρ are both closure operators.) Then we obtain a weakening of classic NI as follows:

$$\forall x_1, x_2 \in \mathbb{V} \,.\, \eta(x_1^{\mathtt{L}}) = \eta(x_2^{\mathtt{L}}) \;\Rightarrow\; \rho(\llbracket P\rrbracket(x_1)^{\mathtt{L}}) = \rho(\llbracket P\rrbracket(x_2)^{\mathtt{L}}) \tag{2}$$

This weakening was first advanced by (Giacobazzi and Mastroeni 2004) where it is termed *narrow (abstract) noninterference (NNI)*. Classic NI is recovered by setting ρ and η to be the identity.

If the attacker cannot access the code, the above observation policy only allows it to understand whether the secret input interferes with the public output or not. Absent code, the attacker cannot understand the manner in which this interference happens. However, given the observation policy, the programmer of the code can analyse the code for vulnerabilities by performing a backwards analysis that computes the maximal relation between secret inputs and public outputs.

Observation policy for a generic semantic policy. We now give an example of how to combine a semantic policy that comprises of a trace-based semantics with an observation policy. In other words we show how we abstract a trace by a state abstraction. Consider the following concrete trace where each state in the trace is represented as the pair $\langle h, l\rangle$.

$$\langle 3,1\rangle \to \langle 2,2\rangle \to \langle 1,3\rangle \to \langle 0,4\rangle \to \langle 0,4\rangle$$

Suppose also that the trace semantics fixes the observation points to be each intermediate state of computation. Now suppose the observation policy is that only the parity (represented by the abstract domain, *Par*) of the public data can be observed. Then the observation of the above trace through *Par* is:

$$\langle 3, \mathit{odd}\rangle \to \langle 2, \mathit{even}\rangle \to \langle 1, \mathit{odd}\rangle \to \langle 0, \mathit{even}\rangle \to \langle 0, \mathit{even}\rangle$$

We can formulate the abstract notion of noninterference on traces by saying that all the execution traces of a program starting from states with different confidential inputs and the same property (say η) of public input, have to provide the same property (say ρ) of public parts of reachable states. Therefore, the general notion of narrow noninterference consists simply in abstracting each state of the computational trace. We thus have

Definition 3.4 [Trace-based NNI] Given a set of observation points $\mathtt{O}$

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash} \,.\, \eta(s_1^{\mathtt{L}}) = \eta(s_2^{\mathtt{L}}) \;\Rightarrow\; \rho(\mathit{post}_P^j(s_1)^{\mathtt{L}}) = \rho(\mathit{post}_P^j(s_2)^{\mathtt{L}})$$

The following example shows the meaning of the observation policy, even if it will not be further considered in the paper.

Example 3.5. Consider the program fragment in Ex. 3.3 together with the semantic policies shown so far. If we consider the semantic policy in (a) and an attacker able only to observe in output the sign of integer variables, i.e., $\eta = \mathit{id}$ and $\rho = \{\varnothing, <0, \geq 0, \top\}$, trivially we have that, for any pair of initial states s_1 and s_2 agreeing on the public part, $\rho(\mathit{post}^5(s_1))^{\mathtt{L}} = (\geq 0) = \rho(\mathit{post}^5(s_2))^{\mathtt{L}}$. Namely, the program is secure. Consider now the semantic policy in (b) and the same observation policy. In this case, noninterference is still satisfied in o_5 but it fails in o_3 and in o_4 since by changing the sign of h_2 we change the sign of respectively l_1 and l_2.

It is worth noting that the output abstraction decides what is observable. In the example above, and in general in the abstract noninterference framework, the abstraction ρ is a property on public data, but it can also be interpreted as a further state projection only on *some* public variables. In other words, in the example we suppose that in all the observable points the attacker can observe *all* the public variables, but we can also suppose that it can only observe some of them, for instance only those modified in the observed point. In the example, we would have that in o_4 only l_2 would be observable. In the following we suppose that the attacker can always observe the whole public state.

3.3. *The protection/declassification policy*

This component of the noninterference policy specifies *what* must be protected or dually, what must be declassified.

For a denotational semantics, classic NI says that nothing must be declassified. Formally, just as in the example in Sec. 1.2, we can say that the property $\top$ has been declassified. From the perspective of an attacker, this means that every secret input has been mapped to $\top$. On the other hand, suppose we want to declassify the parity of the secret inputs. Then we do not care if the attacker can observe any change due to the variation of parity of secret inputs. For this reason, we only check the variations of the output when the secret inputs have the same parity property. Formally, consider an abstract domain ϕ — the declassifier, that is a function that maps any secret input to its corresponding parity. [4] Then a program P satisfies declassified noninterference (DNI) provided

$$\forall x_1, x_2 \in \mathbb{V} \,.\, x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\wedge\; \phi(x_1^{\mathtt{H}}) = \phi(x_2^{\mathtt{H}}) \;\Rightarrow\; [\![P]\!](x_1^{\mathtt{H}}, x_1^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](x_2^{\mathtt{H}}, x_2^{\mathtt{L}})^{\mathtt{L}} \tag{3}$$

This notion of DNI has been advanced several times in the literature, e.g., by (Sabelfeld and Myers 2004); the particular formulation using abstract interpretation is due to (Giacobazzi and Mastroeni 2004) and is further explained in (Mastroeni 2005) where it is termed "declassification by allowing". The generalization of DNI to Def. 3.2 is straightforward. Since we can only protect the secret inputs, we simply add the condition $\phi(s_1^{\mathtt{H}}) = \phi(s_2^{\mathtt{H}})$ to Def. 3.2. In general, we can consider a different declassification policy ϕ_j corresponding to each observation point o_j, $j \in \mathtt{O}$.

Definition 3.6 [Trace-based DNI.] Consider a set of observation points $\mathtt{O}$. Let $\forall j \in \mathtt{O}\ \phi_j$ be the input property declassified in o_j

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash} \,.\, s_1^{\mathtt{L}} = s_2^{\mathtt{L}} \;\wedge\; \phi_j(s_1^{\mathtt{H}}) = \phi_j(s_2^{\mathtt{H}}) \Rightarrow\; post_P^j(s_1)^{\mathtt{L}} = post_P^j(s_2)^{\mathtt{L}}$$

As we can show in the following example, this definition allows a versatile interpretation of the relation between the *where* and the *what* dimensions for declassification. Indeed if we have a unique declassification policy, ϕ, that holds for all the observation points, then it means that $\forall j \in \mathtt{O}. \phi_j = \phi$. On the other hand, if we have explicit declassification, then we have two different choices. We can combine *where* and *what* supposing that the attacker knowledge can only increase. In this case, for any i in $\mathtt{O}$, let ϕ'_i denote the information declassified in the corresponding program point i. Then, the family of declassification policies ϕ_j used in Def. 3.6 is $\phi_j = \sqcap_{i \leq j} \phi'_i$. In other words, in each observation point we declassify not only what is explicitly declassified in that point but also what was declassified previously. For instance, consider the following program fragment:

$$P = \left[\begin{array}{lcr} p_0 \rightarrow & & \leftarrow o_0 \\ & l_1 := declassify(h \geq 0); & \\ & & \leftarrow o_1 \\ & l_2 := declassify(h \leq 0); & \\ & & \leftarrow o_2 \end{array}\right.$$

Then $\phi'_1 = \phi_1 = \{\top, h \geq 0, \varnothing\}$ but in o_2 we have $\phi'_2 = \{\top, h \leq 0, \varnothing\}$ and $\phi_2 = \phi'_1 \sqcap \phi'_2 = \{\top, h \geq 0, h \leq 0, h = 0\varnothing\}$. Hence in the program point o_2 we explicitly declassify only $h \leq 0$, but together with what was previously declassified, we declassify also $h = 0$. We call this kind of declassification *incremental* declassification. On the other hand, we can combine *where* and *what* in a stricter way. Namely, the information is

[4] More precisely, ϕ maps a set of secret inputs to the join of the parities obtained by mapping ϕ to each secret input; the join of *even* and *odd* is $\top$.

declassified only in the particular observation point where it is explicitly declared, but not in the following points. In this case it is sufficient to consider as ϕ_j exactly the property corresponding to the explicit declassification and we call it *localized* declassification. This kind of declassification can be useful when we consider the case where the information obtained by the different observation points cannot be combined, for example if different points are observed by different attackers, as it can happen in the security protocol context.

Example 3.7. Consider the program fragment in Ex. 3.3, with the only difference that the third line is substituted with $l_1 := \mathit{declassify}(h_2)$. Consider the I/O semantic policy in the picture (a), then we don't consider where the declassification is declared, like in delimited release (Sabelfeld and Myers 2004). Indeed, for delimited release the underlying security policy is that h is declassified at the input, and the program point where the declassification actually happens is ignored (Askarov and Sabelfeld 2007a). In other words, declassification of h is visible to the entire program. Hence, $\phi_5(h_1, h_2) = \langle \top(h_1), \mathit{id}(h_2)\rangle \overset{\text{def}}{=} \mathit{id}_{h_2}$, namely we declassify the value of h_2. With this declassification policy noninterference is satisfied. Consider now the semantic policy in picture (b). In this case we have two possibilities. If we consider *incremental* declassification, then we have $\phi_3 = \phi_4 = \phi_5 = \mathit{id}_{h_2}$. Of course in this case the program is trivially secure, since everything about h_2 is released but nothing about h_1 is released: indeed h_1's value is lost in the first line of P due to a destructive update. But, if we consider *local* declassification, then $\phi_3 = \mathit{id}_{h_2}$ but $\phi_4 = \phi_5 = \top$, since in o_4 and in o_5 there are not declassifications. In this case the program is insecure since, as shown in Ex. 3.3, in both o_4 and o_5 we release something about h_2, respectively its value and its parity.

4. $\mathcal{F}$-completeness and noninterference

In the previous section we characterize noninterference in terms of three dimensions: the semantic policy depends on the set of observation points and protection points; the observation policy depends on the property the attacker can observe in these program points; and the protection policy depends on the property we want to protect in the private input (the only protection point we are going to consider in the rest of the paper). Our goal now is to provide a checking technique that is parametric on all these characteristics. While the technique itself will be introduced in the next section, in this section we will study the fundamental components of the technique and their formal justification.

For the semantic policy, we will use weakest precondition semantics: the intuition is that such a semantics models the reverse engineering process that an attacker would perform for deriving private input properties from public outputs. We will rely on this intution for checking (declassified) NI. The formal justification of this choice of semantics rests on the connection between NI and completeness of abstract interpretations (Giacobazzi and Mastroeni 2005) which we now consider.

In the example from the introduction (Sec. 1.2), we illustrated that as expected, the program $l := h_1$ violates classic NI by showing that the backwards completeness equation

(1) does not hold for the program. Equation (1) gives us a way to *dynamically* check whether a program satisfies a confidentiality policy: indeed, we employ the denotational semantics of a program in the process. Can we do this check statically?

We will see presently that static checking involves $\mathcal{F}$-completeness, instead of $\mathcal{B}$-completeness, and the use of weakest preconditions instead of the denotational semantics. With weakest preconditions, (written Wlp_P), equation (1) has the following equivalent reformulation:

$$\mathcal{H} \circ Wlp_P \circ \mathcal{H} = Wlp_P \circ \mathcal{H} \tag{4}$$

Equation (4) says that $\mathcal{H}$ is $\mathcal{F}$-complete for Wlp_P. In other words: consider the abstraction of a concrete input state, X, via $\mathcal{H}$; this yields a set of states where the secret information is abstracted to "any possible value". The equation asserts that $Wlp_P(\mathcal{H}(X))$ is a fixpoint of $\mathcal{H}$, meaning that $Wlp_P(\mathcal{H}(X))$ yields a set of states where each public output is associated with any possible secret input: a further abstraction of the fixpoint (cf., the lhs of equation (4)) yields nothing new. Because no *distinctions among secret inputs* get exposed to an observer, the public output is independent of the secret input. Hence equation (4) asserts classic NI.

The following theorem asserts that the two ways of describing noninterference by means of $\mathcal{B}$- and $\mathcal{F}$-completeness are equivalent.

Theorem 4.1. *$\mathcal{H} \circ [\![P]\!] \circ \mathcal{H} = \mathcal{H} \circ [\![P]\!]$ iff $\mathcal{H} \circ Wlp_P \circ \mathcal{H} = Wlp_P \circ \mathcal{H}$.*

Proof. By (Giacobazzi and Quintarelli 2001, Sec.4) we know that, if f is additive there exists its right adjoint (written f^+) then for any ρ we have $\rho \circ f \circ \rho = \rho \circ f$ iff $\rho \circ f^+ \circ \rho = f^+ \circ \rho$. By (Cousot 2002, Sec. 9) we have that $[\![P]\!]^+ = Wlp_P$. Choosing $\mathcal{H}, [\![P]\!]$ as ρ, f resp., we are done. $\square$

The following example shows how we can interpret in the weakest precondition-based completeness equation when NI holds.

Example 4.2. Consider the program P, let $\mathbb{V}^{\mathtt{H}} = \{0,1\} \times \{0,1\}$, $\mathbb{V}^{\mathtt{L}} = \{0,1\}$.

$$P \stackrel{\text{def}}{=} \left[\begin{array}{ll} \textbf{if } h_1 \neq h_2 & \textbf{then } l := h_1 + h_2 \\ & \textbf{else } l := h_1 - h_2 + 1; \end{array} \right.$$

$$[\![P]\!]: \quad \langle h_1, h_2, l\rangle \mapsto \langle h_1, h_2, 1\rangle \qquad Wlp_P: \quad \begin{cases} \langle h_1, h_2, 1\rangle & \mapsto \quad \{\langle h_1, h_2\rangle\} \times \mathbb{V}^{\mathtt{L}} \\ \langle h_1, h_2, l\rangle & \mapsto \quad \varnothing \qquad l \neq 1 \end{cases}$$

The public output l is always 1, hence P is secure, as the following calculation shows. Given $\mathbb{V}^{\mathtt{H}} \times \{l\} \in \mathcal{H}$, we can prove that $\mathcal{B}$-completeness for $[\![P]\!]$ holds:

$$\mathcal{H}([\![P]\!](\mathbb{V}^{\mathtt{H}} \times \{l\})) = \mathcal{H}(\mathbb{V}^{\mathtt{H}} \times \{1\}) = \mathbb{V}^{\mathtt{H}} \times \{1\} = \mathcal{H}(\langle h_1, h_2, 1\rangle) = \mathcal{H}([\![P]\!](\langle h_1, h_2, l\rangle))$$

$\mathcal{F}$-completeness for Wlp_P holds also:

$$\begin{array}{l} \mathcal{H}(Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{1\})) = \mathcal{H}(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}}) = \mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}} = Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{1\}) \\ \mathcal{H}(Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{l \neq 1\})) = \mathcal{H}(\varnothing) = \varnothing = Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{l \neq 1\}) \end{array}$$

This equality says exactly that the set of all the private inputs leading to a particular observation ($Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{1\})$) is the set of all the private inputs (since $\mathcal{H}$ abstract the private inputs to $\top$). Thus NI holds.

Therefore, in the following, we will use the wlp semantics in order to check noninterference in terms of completeness. Together with completeness we inherit in particular an abstract domain transformer which refines the input abstract domain for inducing completeness (Giacobazzi et al. 2000). In this context, this transformer would refine the input domain for inducing noninterference, and it has been proved (Giacobazzi and Mastroeni 2005) that this input domain characterizes the *maximal* amount of information disclosed by the program semantics. Intuitively, this refinement process for an abstract domain O w.r.t. a function $f : I \longrightarrow O$ adds all the direct images of f to O, i.e., $f(I)$ (Giacobazzi and Quintarelli 2001). For instance, if we consider Ex. 2.1, we have $Wlp(\langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, \{1\}\rangle) = \langle \{1\}, \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}\rangle$; this means that the observation of 1 in output is due to the private input $h_1 = 1$ which is clearly different from $\mathcal{H} \circ Wlp(\langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, 1\rangle) = \langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}\rangle$ (analogously if we observe 0 in output). Hence completeness does not hold, and the refinement process would enrich the abstract domain $\top = \{\mathbb{V}^{\mathtt{H}}\}$ with the elements $\{1\}$, $\{0\}$ and consequently $\{\varnothing\}$ (for being an abstract domain) modelling the fact that we are releasing the exact value of the binary variable h_1.

5. Declassified NI for I/O semantics

We are now ready to explore the connection between completeness and DNI. In preparation, we revisit the example from Sec. 1.2, but now consider that for the program $P \overset{\text{def}}{=} l := h_1$, the security policy allows declassification of h_1. In this case the program *would* be secure. Equation (1) must naturally be modified by "filtering" $\mathcal{H}$ through a declassifier, $\phi : \wp(\mathbb{V}^{\mathtt{H}}) \to \wp(\mathbb{V}^{\mathtt{H}})$, that provides an abstraction of the secret inputs. Let this "filtered $\mathcal{H}$" be written $\mathcal{H}^{\phi} : \wp(\Sigma) \to \wp(\Sigma)$ ($\mathcal{H}^{\phi}$ will be formally defined in Sec. 5.1). Then we require

$$\mathcal{H} \circ [\![P]\!] \circ \mathcal{H}^{\phi} = \mathcal{H} \circ [\![P]\!] \qquad (5)$$

That is, $[\![P]\!]$ applied to a concrete input, x, and $[\![P]\!]$ applied to the abstraction of x *where the* $\mathtt{H}$ *component of* x *has been declassified by* ϕ, both abstract to the same value. It is easy to show that $\mathcal{H}^{\phi}$ is an uco and that $(\mathcal{H}, \mathcal{H}^{\phi})$ is $\mathcal{B}$-complete for $[\![P]\!]$.

As in Sec. 1.2, let X be $\{\langle 0,0,1\rangle\}$. We are interested in ϕ's behavior on $\{\langle 0,0\rangle\}$, because $\{\langle 0,0\rangle\}$ specifies the values of h_1, h_2 in X. We have, $\phi(\{\langle 0,0\rangle\}) = \{\langle 0,0\rangle, \langle 0,1\rangle\}$: ϕ is the *identity* on what must be declassified – we are releasing the exact value of h_1 – but ϕ is $\top$ on what must be protected, which explains why both $\langle 0,0\rangle$ and $\langle 0,1\rangle$ appear. Now $\mathcal{H}^{\phi}(X) = \phi\{\langle 0,0\rangle\} \times X^{\mathtt{L}} = \{\langle 0,0,1\rangle, \langle 0,1,1\rangle\}$ so that $[\![P]\!](\mathcal{H}^{\phi}(X)) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle\}$ and $\mathcal{H}([\![P]\!](\mathcal{H}^{\phi}(X))) = \mathbb{V}^{\mathtt{H}} \times \{0\}$. This is equal to $\mathcal{H}([\![P]\!](X))$. We can show equation (5) for any $X \subseteq \Sigma$; hence $l := h_1$ is secure. Note how ϕ partitions $\wp(\mathbb{V}^{\mathtt{H}})$ into blocks $\{\langle 0,0\rangle, \langle 0,1\rangle\}$ (the range of $\langle 0,0\rangle$ and $\langle 0,1\rangle$) and $\{\langle 1,0\rangle, \langle 1,1\rangle\}$ (the range of $\langle 1,0\rangle$ and $\langle 1,1\rangle$). Intuitively, ϕ permits exposing distinctions *between blocks* at the public output, e.g., between $\langle 0,0\rangle$ and $\langle 1,0\rangle$; in classic NI, ϕ's range is $\top$ and *no* distinctions should be exposed.

5.1. *Modelling declassification*

The discussion in Sec. 4 has not motivated *why* we might want Wlp_P and this is what we proceed to do in the context of declassification. Moreover, declassification is a "property" of the system input, hence it is natural to characterize/verify it with a backward analysis from the outputs towards the inputs, i.e., by means of the wlp semantics of the program.

Consider secrets $h_1, h_2 \in \{0,1\}$ and the declassification policy "at most one of the secrets h_1, h_2 is 1". The policy releases a relation between h_1 and h_2 but not their exact values. Does the program $P \stackrel{\text{def}}{=} l := h_1 + h_2$ satisfy the policy?

Here $\mathbb{V}^{\mathtt{H}} = \{0,1\} \times \{0,1\}$ and the declassifier, ϕ, is defined as: $\phi(\varnothing) = \varnothing$; $\phi\{\langle 0,0\rangle\} = \phi\{\langle 0,1\rangle\} = \phi\{\langle 1,0\rangle\} = \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle\}$ (i.e., we collect together all the elements with the same declassified property) and $\phi\{\langle 1,1\rangle\} = \mathbb{V}^{\mathtt{H}}$; $\phi(X) = \bigcup_{x\in X}(\phi(\{x\}))$. A program that respects the above policy *should not expose the distinctions* between inputs $\langle 0,0\rangle$, $\langle 0,1\rangle$ and $\langle 1,0\rangle$ at the public output. But it *is* permissible to expose the distinction between $\langle 1,1\rangle$ and any pair from the partition block $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle\}$, because this latter distinction is supported by the policy. Does P expose distinctions it should not?

To answer, we consider $Wlp_P(l = a)$, where a is some generic output value. Why Wlp? Because then we can statically simulate the kind of analysis an attacker would do for obtaining initial values of (or initial relations among) secret information. Why $l = a$? Because this gives us the *most general Wlp, parametric on the output value* (following (Gorelick 1975). Now, note that $Wlp_P(l = a) = (h_1 + h_2 = a)$; let $W_a \stackrel{\text{def}}{=} (h_1 + h_2 = a)$. Because $a \in \{0,1\}$, we have $W_0 = (h_1 + h_2 = 0)$. This allows the attacker to solve for h_1, h_2: $h_1 = 0, h_2 = 0$. Thus when $l = 0$, a distinction, $\{\langle 0,0\rangle\}$, in the partition block, $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle\}$ gets exposed. Hence the program does not satisfy the policy.

So consider a declassified confidentiality policy, and model the declassified information by means of the abstraction ϕ, of the secret inputs, which collects together all the elements with the same property, declassified by the policy. Let $\mathcal{H}^\phi : \wp(\Sigma) \to \wp(\Sigma)$ be the corresponding abstraction function. Let $X \in \wp(\Sigma)$ be a concrete set of states and let $X^{\mathtt{L}}$ be the $\mathtt{L}$ slice of X. Consider any $l \in X^{\mathtt{L}}$. Define set $X_l \stackrel{\text{def}}{=} \{h \in \mathbb{V}^{\mathtt{H}} \mid \langle h, l\rangle \in X\}$; i.e., given an l, X_l contains all the $\mathtt{H}$ values associated with l in X. Then the "declassified" abstract domain, $\mathcal{H}^\phi(X)$, corresponding to X is defined as

$$\mathcal{H}^\phi(X) = \bigcup_{l\in X^{\mathtt{L}}} \phi(X_l) \times \{l\}$$

Note that the domain, $\mathcal{H}$, for ordinary noninterference is the instantiation of $\mathcal{H}^\phi$, where ϕ maps any set to $\top$. The analogue of equation (5),

$$\mathcal{H}^\phi \circ Wlp_P \circ \mathcal{H} = Wlp_P \circ \mathcal{H} \tag{6}$$

asserts that $(\mathcal{H}^\phi, \mathcal{H})$ is $\mathcal{F}$-complete for Wlp_P. For example, $\mathcal{F}$-completeness fails for the program P above. With $X = \langle 0,0,0\rangle$, we have $\mathcal{H}(X) = \mathbb{V}^{\mathtt{H}} \times \{0\}$ and $Wlp_P(\mathcal{H}(X)) = \{\langle 0,0,0\rangle\}$. But $\mathcal{H}^\phi(Wlp_P(\mathcal{H}(X))) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \langle 1,0,0\rangle\} \supset Wlp_P(\mathcal{H}(X))$.

We are now in a position, via Theorem 5.1 below, to connect $\mathcal{H}^\phi$ to DNI: the only caveat is that ϕ must *partition* the input abstract domain, i.e., $\forall x.\phi(x) = \{y \mid \phi(x) = \phi(y)\}$. The intuition behind partitioning is that ϕ's image on singletons is all we need for deriving the property of any possible set.

Theorem 5.1. Consider a partitioning ϕ. Then P satisfies noninterference declassified by ϕ iff $\mathcal{H} \circ [\![P]\!] \circ \mathcal{H}^\phi = \mathcal{H} \circ [\![P]\!]$.

Proof. Let us prove first that if the program satisfies declassified noninterference then the abstract domain is complete. Note that by hypothesis we have that

$$x_1^{\mathtt{L}} = x_2^{\mathtt{L}} \;\wedge\; \phi(x_1^{\mathtt{H}}) = \phi(x_2^{\mathtt{H}}) \;\Rightarrow\; [\![P]\!](x_1)^{\mathtt{L}} = [\![P]\!](x_2)^{\mathtt{L}}$$

Note that all the functions involved in the equation are additive, hence also their composition is additive. This means that we can prove the equality simply on the singletons. Moreover, since ϕ is partitioning, we have that $\phi(x) = \left\{\; y \;\middle|\; \phi(x) = \phi(y) \;\right\}$, hence for any $z \in \phi(x^{\mathtt{H}})$ we have $\phi(z) = \phi(x^{\mathtt{H}})$, and therefore $[\![P]\!](\phi(x^{\mathtt{H}}), x^{\mathtt{L}})^{\mathtt{L}} = \bigcup_{z\in\phi(x^{\mathtt{H}})} [\![P]\!](z, x^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](x^{\mathtt{H}}, x^{\mathtt{L}})^{\mathtt{L}}$, due to the NI hypothesis. Hence the following equalities hold:

$$\begin{aligned} \mathcal{H} \circ [\![P]\!] \circ \mathcal{H}^\phi(x) &= \mathcal{H} \circ [\![P]\!](\phi(x^{\mathtt{H}}), x^{\mathtt{L}}) \\ &= \mathcal{H} \circ [\![P]\!](x) \text{ for what we proved above.} \end{aligned}$$

On the other side, completeness says that for any x we have $[\![P]\!](\phi(x^{\mathtt{H}}), x^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](x^{\mathtt{H}}, x^{\mathtt{L}})^{\mathtt{L}}$, which trivially implies noninterference declassified by ϕ. $\square$

Together with a straightforward generalisation of Theorem 4.1 to different input and output abstractions, we are led to

Corollary 5.2. Consider a partitioning ϕ. Then P satisfies noninterference declassified by ϕ iff $\mathcal{H}^\phi \circ Wlp_P \circ \mathcal{H} = Wlp_P \circ \mathcal{H}$, i.e., $(\mathcal{H}^\phi, \mathcal{H})$ is $\mathcal{F}$-complete for Wlp_P.

The equality in the corollary asserts that *nothing more is released* by the *Wlp* than what is already released by ϕ. If $\mathcal{F}$-completeness did not hold, but $(\mathcal{H}^\phi, \mathcal{H})$ was merely sound, then $\mathcal{H}^\phi \circ Wlp_P \circ \mathcal{H} \sqsupseteq Wlp_P \circ \mathcal{H}$. In this case *Wlp* (i.e., the rhs) releases *more information* (technically: is more concrete) than what is declassified (i.e., the lhs). Our goal is not only to check whether a program satisfies a particular confidentiality policy, but also to find the public observations that may breach the confidentiality policy and also the associated secret that each offending observation reveals. Consider, for example, the following program (Darvas et al. 2005) where $l, h \in \mathsf{Nats}$.

$$P \overset{\text{def}}{=} \textbf{while } (h > 0) \textbf{ do } (h := h - 1; l := h) \textbf{ endw}$$

If we observe $l = 0$ at output, all we can say about input h is $h \geq 0$. But with output observation $l \neq 0$, we can deduce $h = 0$ in the input: the loop must not have been executed.

Because *Wlp* relates the observed (public) output to the private (secret) inputs, therefore, from the final observation we can derive the exact secret which is released by that observation in the following manner: (a) Compute wlp w.r.t. each observation obtaining a most general predicate on the input states. (b) Check whether the states described by the wlp are "more abstract", i.e., do not permit more distinctions of secret inputs than those permitted by the policy. If so, there is no breach. The following examples show how we can use weakest precondition semantics for checking declassified policies.

Example 5.3. Consider the following code (Sabelfeld and Myers 2004)

$$P \overset{\text{def}}{=} h := h \ mod \ 2; \ \textbf{if} \ h = 0 \ \textbf{then} \ (h := 0; l := 0) \ \textbf{else} \ (h := 1; l := 1);$$

Let $\mathbb{V}^{\mathtt{H}} = \mathsf{Nats} = \mathbb{V}^{\mathtt{L}}$. Suppose we wish to declassify the test, $h = 0$. Then $\phi(\{0\}) = \{0\}$ and $\phi(\{h\}) = \mathsf{Nats} \smallsetminus \{0\}$. Thus $\{\{h \mid h \neq 0\}, \{0\}\}$ is the partition induced by ϕ on $\mathbb{V}^{\mathtt{H}}$ and we obtain $\mathcal{H}^{\phi} = \{\varnothing, \top\}^{5} \cup \{\{h \mid h \neq 0\} \times \mathbb{V}^{\mathtt{L}}\} \cup \{\{0\} \times \mathbb{V}^{\mathtt{L}}\}$.
Let $H_a \overset{\text{def}}{=} \mathbb{V}^{\mathtt{H}} \times \{a\}$. Consider now the wlp of the program, $Wlp_P(l = a)$, where $a \in \mathbb{V}^{\mathtt{L}}$.

$$P = \left[\begin{array}{lll} p_0 \to & \{(a = 0, \ h \ mod \ 2 = 0) \ \vee \ (a = 1, \ h \ mod \ 2 = 1)\} & \leftarrow o_0 \\ & h := h \ mod \ 2; & \\ & \{(a = 0, \ h = 0) \ \vee \ (a = 1, \ h = 1)\} & \\ & \textbf{if} \ (h = 0) \ \textbf{then} \ (h := 0; l := 0) \ \textbf{else} \ (h := 1; l := 1) & \\ & \{l = a\} & \leftarrow o_2 \end{array} \right.$$

Thus, Wlp maps output set of states H_0 to the input states $\{\langle h, l \rangle \mid h \ mod \ 2 = 0, l \in \mathbb{V}^{\mathtt{L}}\}$. But this state is not more abstract than the state, $\{h \mid h \neq 0\} \times \mathbb{V}^{\mathtt{L}}$, specified by $\mathcal{H}^{\phi}$: it distinguishes, e.g., $(8, 1)$ from $(7, 1)$ - a distinction not permitted under the policy. Indeed, consider two runs of P with initial values 8 and 7 of h and 1 for l; $\phi(8) = \phi(7)$; yet we get two distinct output values of l.

Example 5.4. Consider $P \overset{\text{def}}{=} \textbf{if} \ (h \geq k) \ \textbf{then} \ (h := h - k; l := l + k) \ \textbf{else skip}$ (Sabelfeld and Myers 2004), where $l, k : \mathtt{L}$. Consider $H_{a,b} \overset{\text{def}}{=} \{\langle h, l, k \rangle \mid h \in \mathbb{V}^{\mathtt{H}}, \ l = a, \ k = b\}$. Suppose the declassification policy is $\top$, i.e., nothing has to be released.

$$P = \left[\begin{array}{lll} p_0 \to & \{(h \geq b, \ l = a - b, \ k = b) \ \vee (h < b, \ l = a, \ k = b)\} & \leftarrow o_0 \\ & \textbf{if} \ (h \geq k) \ \textbf{then} \ (h := h - k; l := l + k) \ \textbf{else skip} & \\ & \{l = a, \ k = b\} & \leftarrow o_1 \end{array} \right.$$

$$Wlp_P : H_{a,b} \mapsto \ \{\langle h, a - b, b \rangle \mid h \geq b\} \ \cup \ \{\langle h, a, b \rangle \mid h < b\}$$

In this case, we can say that the program does not satisfy the security policy. In fact, in presence of the same public inputs we can distinguish between values h greater than the initial value of k, and lower than this value.

6. Declassified NI on traces

In this section, we show that we can use the weakest precondition-based approach seen in the previous section in order to check declassification policies w.r.t. trace-based DNI (Def. 3.6). Before understanding how our approach works on trace-based noninterference by means of examples, we provide the formal justification for the technique we use on traces.

[5] These elements are necessary for being an upper closure operator.

Since the denotational semantics is the *post* of a transition system where all traces are two-states long – because they are input-output states — a straightforward generalization of the completeness reformulation, in order to cope also with trace-based NI (Def. 3.2), can be immediately obtained. Theorem 6.1 below shows the connection, also for traces, between completeness and noninterference.

Theorem 6.1. Let $\langle \Sigma, \rightarrow_P \rangle$ be the transition system for a program P, and $\mathtt{O}$ a set of observation points. Then, noninterference as in Def. 3.2, i.e.,

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash}.\, s_1 =_{\mathtt{L}} s_2 \;\Rightarrow\; \mathit{post}^j_P(s_1) =_{\mathtt{L}} \mathit{post}^j_P(s_2)$$

holds iff $\forall j \in \mathtt{O}.\, \mathcal{H} \circ \mathit{post}^j_P \circ \mathcal{H} = \mathcal{H} \circ \mathit{post}^j_P$.

Proof. In the following we will simply denote by post^j the function post^j_P. Consider the property $\forall s_1, s_2 \in \Sigma_{\vdash}.\, s_1 =_{\mathtt{L}} s_2 \;\Rightarrow\; \mathit{post}^j(s_1) =_{\mathtt{L}} \mathit{post}^j(s_2)$, i.e., $\mathcal{H}(\mathit{post}^j(s_1)) = \mathcal{H}(\mathit{post}^j(s_2))$ for all $s_1 =_{\mathtt{L}} s_2$. Hence, for all s_1, s_2 such that $\mathcal{H}(s_1) = \mathcal{H}(s_2)$ we have that $\mathcal{H}(\mathit{post}^j(s_1)) = \mathcal{H}(\mathit{post}^j(s_2))$. By addtivity of $\mathcal{H}$ this implies that for all s and for all $s' \in \mathcal{H}(s)$,

$$\mathcal{H}(\mathit{post}^j(s')) = \mathcal{H}(\mathit{post}^j(s))$$

Again by addtivity of $\mathcal{H}$ we obtain

$$\mathcal{H}(\mathit{post}^j(\mathcal{H}(s))) = \mathcal{H}(\mathit{post}^j(s))$$

On the other side, if we have completeness, i.e., $\mathcal{H}(\mathit{post}^j(\mathcal{H}(s))) = \mathcal{H}(\mathit{post}^j(s))$ we can prove that for all s_1, s_2 such that $s_1 =_{\mathtt{L}} s_2$, i.e., such that $\mathcal{H}(s_1) = \mathcal{H}(s_2)$, we have $\mathcal{H}(\mathit{post}^j(s_1)) = \mathcal{H}(\mathit{post}^j(s_2))$, hence the semantics are the same from the public point of view. Indeed, for all s and for all $s' \in \mathcal{H}(s)$ (that is $\mathcal{H}(s) = \mathcal{H}(s')$) we have by completeness that $\mathcal{H}(\mathit{post}^j(s)) = \mathcal{H}(\mathit{post}^j(\mathcal{H}(s)))$ and that $\mathcal{H}(\mathit{post}^j(\mathcal{H}(s'))) = \mathcal{H}(\mathit{post}^j(s'))$. But clearly, by hypothesis on s and s' we have $\mathcal{H}(\mathit{post}^j(\mathcal{H}(s))) = \mathcal{H}(\mathit{post}^j(\mathcal{H}(s')))$. Hence, we have the thesis, i.e., $\mathcal{H}(\mathit{post}^j(s)) = \mathcal{H}(\mathit{post}^j(s'))$. □

This theorem implies that we can characterize NI as a family of completeness problems also when a malicious attacker can potentially observe the whole trace semantics, namely when we deal with trace-based NI. As corollary of the theorem above, together with Theorem 5.1, we have the following result.

Corollary 6.2. Let $P \in \textsc{Imp}$, $\langle \Sigma, \rightarrow_P \rangle$ the corresponding transition system, $\mathtt{O}$ a set of observation points, and $\phi \in \mathit{uco}(\wp(\mathbb{V}^{\mathtt{H}}))$ a partitioning closure. Then, noninterference as in Def. 3.6, i.e.,

$$\forall j \in \mathtt{O}.\, \forall s_1, s_2 \in \Sigma_{\vdash}.\, s_1 =_{\mathtt{L}} s_2 \wedge \phi_j(s_1^{\mathtt{H}}) = \phi_j(s_2^{\mathtt{H}}) \Rightarrow\; \mathit{post}^j_P(s_1) =_{\mathtt{L}} \mathit{post}^j_P(s_2)$$

holds iff $\forall j \in \mathtt{O}.\, \mathcal{H} \circ \mathit{post}^j_P \circ \mathcal{H}^{\phi} = \mathcal{H} \circ \mathit{post}^j_P$.

Moreover, let us denote as $\widetilde{\mathit{pre}}^j$ the adjoint map of post^j (noting that, by adjoint relation the function $\widetilde{\mathit{pre}}^j$ is a weakest precondition of the corresponding function post^j) in the same transition system, then the completeness equation can be rewritten as $\mathcal{H} \circ \widetilde{\mathit{pre}}^j \circ \mathcal{H} = \widetilde{\mathit{pre}}^j \circ \mathcal{H}$. In the NI context, this means that there is no leakage of information. In particular, for *declassification*, if $\mathcal{H}^{\phi} \circ \widetilde{\mathit{pre}}^j_P \circ \mathcal{H} = \widetilde{\mathit{pre}}^j_P \circ \mathcal{H}$ holds, there is no need to further declassify secret information via refinement, even if we suppose that the attacker can observe every intermediate step of computation.

These results say that, in order to check DNI on traces, we would have to make an analysis for each observable program point j, combining, afterwards, the information disclosed. In order to make only one iteration on the program even when dealing with traces, our basic idea is to combine the weakest precondition semantics, computed at each observable point of the execution, together with the observation of public data made at the particular observation point.

We will describe our approach on our running example, Example 6.3, where o_i and p_i denote respectively the observation and the protection points of the program P.

Example 6.3.

$$P = \left[\begin{array}{lll} p_0 \rightarrow & \{h_2 \ mod\ 2 = a\} & \leftarrow o_0 \\ & h_1 := h_2; & \\ & h_2 := h_2\ mod\ 2; & \\ & l_1 := h_2; & \\ & h_2 := h_1 & \\ & l_2 := h_2; & \\ & l_2 := l_1 & \\ & \{l_1 = l_2 = a\} & \leftarrow o_6 \end{array}\right.$$

In this example, the observation in output of l_1 and l_2 allows us to derive the information about the parity of the secret input h_2.

6.1. *Localized declassification*

In trace-based NI we underline how we can easily describe the strong relation between the observation point and the corresponding declassification policy. This becomes particularly useful in the presence of explicit declassifications, since it allows a precise characterization of the relation between the *where* and the *what* dimensions of noninterference policies. The idea is to track (by using the wlp computation) the information disclosed in each observation point till the beginning of the computation, and then to compare it with the corresponding declassification policy. The next example shows how we track the information disclosed in each observable program point. We consider as the set of observable program points, `O`, the same set used for gradual release, namely, the program points corresponding to low events. However, in general, our technique allows `O` to be any set of program points. When there is more than one observation point, we will use the notation $[\Phi]^{O}$ to denote that the information described by the assertion Φ can be derived from the set of observation points in O.

Example 6.4.

$$P = \left[\begin{array}{lll}
p_0 \rightarrow & \qquad \{[h_2 = b]^{o_5}, [h_2 \ \mathit{mod}\ 2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \leftarrow o_0 \\
 & h_1 := h_2; & \\
 & \qquad \{[h_1 = b]^{o_5}, [h_2 \ \mathit{mod}\ 2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \\
 & h_2 := h_2 \ \mathit{mod}\ 2; & \\
 & \qquad \{[h_1 = b]^{o_5}, [h_2 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \\
 & l_1 := h_2; & \\
 & \qquad \{[h_1 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}, [l_2 = c]^{o_3}\} & \leftarrow o_3 \\
 & h_2 := h_1 & \\
 & \qquad \{[h_2 = b]^{o_5}, [l_1 = a]^{o_5,o_6}\} & \\
 & l_2 := h_2; & \\
 & \qquad \{[l_1 = a]^{o_5,o_6}, [l_2 = b]^{o_5}\} & \leftarrow o_5 \\
 & l_2 := l_1 & \\
 & \qquad \{l_1 = l_2 = a\} & \leftarrow o_6
\end{array}\right.$$

For instance, at observation point o_5, $[l_1 = a]^{o_5,o_6}$ is obtained either via wlp calculation of $l_2 := l_1$ from o_6, or via the direct observation of public data in o_5, while $[l_2 = b]^{o_5}$ — where b is an arbitrary symbolic value — is only due to the observation of the value l_2 in o_5. The assertion in o_3 — $[h_1 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}$ — is obtained by computing the wlp semantics $\mathit{Wlp}(h_2 := h_1, ([h_2 = b]^{o_5}, [l_1 = a]^{o_3,o_5,o_6}))$, while $[l_2 = c]^{o_3}$ is the observation in o_3. Similarly, we can derive all the other assertions.

It is worth noting, that the attacker is more powerful than the one considered in Example 6.3. In fact, in this case, the possibility of observing l_2 in the program point o_5 allows to derive the exact (symbolic) value of h_2 ($h_2 = b$ where b is the value observed in o_5). This was not possible in Example 6.3 based on simple input-output, since the value of l_2 was lost in the last assignment.

Now, we can use the information disclosed in each observation point, characterized by computing the wlp semantics, in order to check if the corresponding declassification policy is satisfied or not. This is obtained simply by comparing the abstraction modelling the declassification with the state abstraction corresponding to the information released in the lattice of abstract interpretations. We explain the idea in Example 6.5.

Example 6.5. Consider the program in Example 6.4, with the only difference that in o_3 the statement $l_1 := h_2$ is substituted with $l_1 := \mathit{declassify}(h_2)$. In this case, the corresponding declassification policy is $\phi_3 = id_{h_2}$. Now we can compare this policy with the private information disclosed in o_3, which is $h_2 \ \mathit{mod}\ 2 = a$ (h_2's parity) and therefore we conclude that the declassification policy in o_3 is satisfied because parity is more abstract than identity. Nevertheless, the program releases the information $h_2 = b$ in the program point o_5. This means that the security of the programs depends on the kind of localized declassification we consider. For incremental declassification the program is secure since the release is licensed by the previous declassification since the observation point o_5, where we release information corresponds to the declassification policy $\phi_5 = \phi_3 = id_{h_2}$, while for localized declassification the program is insecure since there isn't a corresponding declassification policy at o_5 that licenses the release.

Multiple declassifications. Consider the following example with multiple declassifications. Suppose the attacker can observe any step, but only the secret input needs to be protected. In this case we can see how the attacker can combine, at the protection point, different information (some obtained by *Wlp* and others obtained by declassification) obtaining more information than what is explicitly declassified.

Example 6.6.

$$P = \left[\begin{array}{lll} p_0 \to & \{[h_1 - h_2 = a]^{o_3}, [l_2 = d]^{o_1}, [h_1 + h_2 = b]^{o_1,o_2,o_3}\} & \leftarrow o_0 \\ & \quad \Rightarrow (h_1 = (a+b)/2, h_2 = (b-a)/2) & \\ & l_1 := \mathit{declassify}^{o_1}(h_1 + h_2); & \\ & \{[h_1 - h_2 = a]^{o_3}, [l_1 = b = c]^{o_1,o_2,o_3}, [l_2 = d]^{o_1}\} & \leftarrow o_1 \\ & l_2 := l_1; & \\ & \{[l_2 = b]^{o_2,o_3}, [h_1 - h_2 = a]^{o_3}, [l_1 = c]^{o_2}\} & \leftarrow o_2 \\ & l_1 := \mathit{declassify}^{o_3}(h_1 - h_2); & \\ & \{l_1 = a, l_2 = b\} & \leftarrow o_3 \end{array}\right.$$

This program is not secure, since from the conjunction of two declassified conditions we can further deduce information about the exact (symbolic) values of the secrets. That is, the information released is *id*, which is more concrete than both $\phi_1 = \{\, \{\, \langle h_1, h_2\rangle \mid h_1 + h_2 = n \,\} \mid n \in \mathbb{N} \,\}$ and $\phi_3 = \{\, \{\, \langle h_1, h_2\rangle \mid h_1 - h_2 = n \,\} \mid n \in \mathbb{N} \,\}$.

6.2. *Localized declassification vs Gradual Release*

In this section we compare the idea introduced above with *gradual release* (Askarov and Sabelfeld 2007a) — the first notion introduced for dealing with the *where* dimension of declassification. In gradual release, the only permitted leakages are at the points of declassification and the definition characterizes the *knowledge* that can be deduced from the declassification. Suppose o_1, o_2 are two consecutive declassification points. Then gradual release ensures that the knowledge at o_1 — together with the knowledge of the declassification at o_1 remains constant till the declassification at o_2 happens.

Definition 6.7 (Askarov and Sabelfeld 2007a) A program P satisfies gradual release if for any initial state $s \in \Sigma$ and for any $\sigma^l \in L(P, s) \overset{\text{def}}{=} \{\, \sigma^l \mid \sigma^l_0 = s \,\}$[6] we have that: Let $|\sigma^l| = n$ and let d_j be the indexes of the states obtained after executing an explicit declassification

$$\forall i, j.\ 1 \le i \le n \wedge i \ne d_j \;\Rightarrow\; K_{\downarrow}(P, \sigma_{\overline{i-1}}) = K_{\downarrow}(P, \sigma_{\overline{i}})$$

where $\sigma_{\overline{i}}$ is the prefix of σ with length i, $K_{\downarrow}(P, \sigma) \overset{\text{def}}{=} \{\, \tau_0 \mid \tau \text{ trace s.t. } \sigma^l = \tau^l \,\}$ and τ_0 denotes the initial state of the trace τ.

From Def. 6.7 we note that L makes a *forward* analysis by computing the observed traces starting from a given initial state, while the knowledge K makes a *backward* analysis by

[6] σ^l denotes the projection of the trace σ only on the low events (low assignment, declassification and termination).

computing all the possible input states which can be the initial ones of the observed trace. In other words, L provides the real observations made by an attacker for a given input state, K provides the information that the attacker can derive from this observation. This approach corresponds exactly to the weakest precondition semantics approach proposed in Sec. 5. Let us see how gradual release works and how it can be compared with our approach on some examples.

Example 6.8.

$$P_1 \overset{\text{def}}{=} \left[\begin{array}{lll} p_0 \rightarrow & & \leftarrow o_0 \\ & h_1 := h_2; & \\ & h_2 := h_2 \ mod\ 2; & \\ & l_1 := \mathit{declassify}(h_2); & \\ & & \leftarrow o_3 \\ & h_2 := h_1; & \\ & l_2 := h_2; & \\ & & \leftarrow o_5 \end{array}\right.$$

The low trace of computation is

$$\langle l_1, l_2\rangle \rightarrow \langle h_2 \ mod\ 2, l_2\rangle \rightarrow \ \langle h_2 \ mod\ 2, h_2\rangle$$

Starting from the observable portion, $\langle l_1, l_2\rangle$, of the initial state we discover, in the final state, the parity of h_2 and h_2's value.

Example 6.8 is insecure according to gradual release for the following argument. Although there is a change of knowledge at o_3, owing to the release of h_2's parity into l_2, this release is licensed by the *declassify* statement. However, in state o_5 there is again a change of knowledge due to the flow of h_2 into l_2. But this change of knowledge is *not* licensed by any explicit declassification.

The above argument is the same one we made for Example 6.5; moreover, the *Wlp* calculation allowed us to find the information released by the program.

Note, however, that the example is secure according to delimited release, because we know at the *start* of the program that the initial value of h_2 is declassified.

Askarov and Sabelfed note that the notion of gradual release is not exactly a *what* policy, because it only considers the presence of declassification statements and not *what* information is declassified (Askarov and Sabelfeld 2007a). The two notions, delimited release and gradual release, are incomparable. The example above gives a scenario where delimited release holds but gradual release does not. Similarly gradual release may hold while delimited release does not as in the following example from (Askarov and Sabelfeld 2007a).

Example 6.9.

$$P_2 \overset{\text{def}}{=} \left[\begin{array}{lll} p_0 \rightarrow & & \leftarrow o_0 \\ & h_1 := h_1 \ mod\ 2; & \\ & l_1 := \mathit{declassify}(h_1 = 0); & \\ & & \leftarrow o_2 \end{array}\right.$$

The low trace is $\langle l_1, l_2\rangle \rightarrow \langle h_1 \ mod\ 2, l_2\rangle$. In this case we are releasing the parity since, just before the declassification statement h_2 takes its parity, but explicitly we are declassifying the property of being equal to (or different from) 0. Hence, as is shown for a similar example in (Sabelfeld and Myers 2004), this program does not satisfy delimited release. But it *does* satisfy gradual release since all the released information is leaked in the declassification point, and so we never check it.

In contrast to gradual release, our approach considers the *what* dimension. Thus we can realize whether what is released is *more* than what is declassified — even if the information is released *only* in the declassification point. Let us denote by ϕ_2 the property corresponding to the information declassified in the program, which corresponds to the information of being equal or not to 0.

$$P_2 \overset{\text{def}}{=} \left[\begin{array}{llr} p_0 \rightarrow & \quad\{h_1\ mod\ 2 = a\} & \leftarrow o_0 \\ & h_1 := h_1\ mod\ 2; & \\ & \quad\{(h_1 = 0) = a\} & \\ & l_1 := declassify(h_1 = 0); & \\ & \quad\{l_1 = a, l_1 = b\} & \leftarrow o_2 \end{array}\right.$$

The only information that flows comes through a declassification, but the information released is the parity while the declassification is ϕ_2. Because these properties are incomparable, the program is not secure in our approach. Again, this example shows how our approach can combine the what and the where dimension.

6.3. *Localized delimited release*

Localized delimited release (Askarov and Sabelfeld 2007b) is a notion that *combines* the *what* and the *where* dimensions of declassification. However, as the authors note, this combination may lose *monotonicity of release* (Sabelfeld and Sands 2007): the addition of declassification annotations to the code, can turn a secure program into an insecure one. In particular, the problem arises when there are computations where the declassification statement is not reached, and others where it is executed. Consider the following example from (Askarov and Sabelfeld 2007b).

Example 6.10.

$$P = \left[\begin{array}{llr} p_0 \rightarrow & & \leftarrow o_0 \\ & h' := 0; & \\ & \textbf{if } h \textbf{ then } l := declassify(h') \textbf{ else } l := 0; & \\ & & \leftarrow o_2 \end{array}\right.$$

This program is clearly intuitively secure, because no information about secrets is released. *Without* the declassify statement localized delimited release recognizes this program as secure(Askarov and Sabelfeld 2007b); however, *with* the declassification statement in the if branch, the program is labelled as insecure. This happens because the definition of localized release treats declassification w.r.t. initial states, ignoring the possibility that a declassification may become harmless in the current state (as declassifying

0 in the example). If we apply our technique then we obtain the following analysis recognizing the program as secure, since no information is derived about h from the observation of l.

$$P = \left[\begin{array}{llr} p_0 \rightarrow & \quad \{((h=1, a=0) \;\vee\; ((h=0, a=0)\} \;\Leftrightarrow\; \mathit{true} & \leftarrow o_0 \\ & h' := 0; & \\ & \quad \{(h=1, h'=a) \;\vee\; (h=0, a=0)\} & \\ & \textbf{if } h \textbf{ then } l := \mathit{declassify}(h') \textbf{ else } l := 0; & \\ & \quad \{l = a\} & \leftarrow o_2 \end{array}\right.$$

In general, the problem with localized delimited release is that it seems to be a strong form of delimited release that may add too many false negatives.

7. Deriving counterexamples and refining declassification policies

In this section we show how to generate the set of all possible counterexamples when DNI fails by demonstrating the corresponding set of all possible pairs of input states that break DNI. Note that, the difference induced by the semantic policy in DNI is only in the number of tests — one test for each observation point — that we have to perform for each pair of input states agreeing on the public part. This means that, for each observation point, the relation between observation and declassification does not depend on the semantics and can be treated independently. Hence, in the following we show how we can characterize, for each observation point, the counterexamples to the corresponding declassification policy, when it is not satisfied.

We have advanced the thesis that noninterference is a completeness problem in abstract interpretation. (Ranzato and Tapparo 2005) studied completeness from another point of view. They show the strong connection existing between completeness and the notion of stability, in particular when dealing with partitioning closures (Mastroeni 2008). In other words, there exists a correspondence between completeness and absence of *unstable* elements of a closure w.r.t. a function f: Given a partition $\Pi \subseteq \wp(C)$ and a function $f : \wp(C) \longrightarrow \wp(C)$, an element $X \in \Pi$ is *stable* for f with respect to $Y \in \Pi$ if $X \subseteq f(Y)$ or $X \cap f(Y) = \varnothing$; otherwise X is said to be *unstable*. It is clear that in our declassification context, where we partition the input by $\mathcal{H}^{\phi}$ and the output by $\mathcal{H}$, we don't have the same partition in input and output. This means that we have to consider a slight generalization of the stability notion where the constraint about the equality between the input and the output partition is relaxed (Mastroeni 2008). In particular, if $f : \mathtt{I} \longrightarrow \mathtt{O}$, and $\Pi_{\mathtt{I}}$ and $\Pi_{\mathtt{O}}$ partition respectively $\mathtt{I}$ and $\mathtt{O}$, then $\Pi_{\mathtt{O}}$ is stable w.r.t. f and $Y \in \Pi_{\mathtt{I}}$ if $\forall X \in \Pi_{\mathtt{O}}$ we have $X \subseteq f(Y)$ or $X \cap f(Y) = \varnothing$. The understanding of completeness in terms of stability guarantees that if an abstract domain is not complete then there exist at least two of its elements which are not stable. In our context, f, is *Wlp*; the element for which we want to check stability is a set of secret inputs in the partition of $\mathbb{V}^{\mathtt{H}}$ induced by the declassifier, ϕ; and the element against which we check stability (Y in the definition) is the particular output observation (e.g., $l = a$). In general, the particular output will be the set L of all the public data observed in all the different observable points, i.e., $L = \{L_i \mid L_i \text{ observation in } o_i\}$. For instance, in Ex. 3.3, if the attacker observes I/O, namely if $\mathtt{O} = \{6\}$, then $L = \{l_1 = l_2 = a\}$ is the particular output observation, against which we check the declassification policy $\phi = \phi_3 = id_{h_2}$. Instead, if the attacker can observe all the low events ($\mathtt{O} = \{3, 5, 6\}$), then $L = \{L_3, L_5, L_6\} = \{[l_1 = l_2 = a]^{o_6}, [l_2 = b]^{o_5}, [l_2 = c]^{o_3}\}$ (we consider only the relevant observations) is the set of public observations against which we check the corresponding declassification policies ϕ_3, ϕ_5 and ϕ_6.

Proposition 7.1. Let $\mathtt{O}$ be the set of observable points and let, $\forall i \in \mathtt{O}$, ϕ_i be the corresponding declassified property. For each ϕ_i, unstable elements of $\mathcal{H}^{\phi_i}$ w.r.t. the output partition induced by o_i, $i \in \mathtt{O}$ provide counterexamples to ϕ_i.

Proof. Let us denote simply by ϕ any of the declassification policies ϕ_i. Suppose that $\exists L \subseteq \mathbb{V}^{\mathtt{L}}$ (observation in o_i) such that (the input states described by) $Wlp(H_L) \notin \mathcal{H}^{\phi}$. Then there exist $x \in Wlp(H_L)$, and $h \in \phi(x^{\mathtt{H}})$ such that $\langle h, x^{\mathtt{L}}\rangle \notin Wlp(H_L)$. Note that $(\phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\}) \cap Wlp(H_L) \neq \varnothing$ since x is in both; and, $\phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\} \not\subseteq Wlp(H_L)$ since we have that $\langle h, x^{\mathtt{L}}\rangle \in \phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\}$ and $\langle h, x^{\mathtt{L}}\rangle \notin Wlp(H_L)$. Hence, the abstract domain $\mathcal{H}^{\phi}$ is not *stable*. To find a counterexample consider $h_1 \in \phi(x^{\mathtt{H}}) \smallsetminus \{k \mid \langle k, x^{\mathtt{L}}\rangle \in Wlp(H_L)\}$ and $h_2 \in \phi(x^{\mathtt{H}}) \cap \{k \mid \langle k, x^{\mathtt{L}}\rangle \in Wlp(H_L)\}$. The latter set is obtained by wlp for the output observation L, hence any of its elements, e.g., h_2, leads to the observation L, while all the elements outside the set, e.g., h_1, cannot lead to L. But, both h_1 and h_2 are in the same element of ϕ, hence they form a counterexample to the declassification policy ϕ. □

Next example shows how we can use the above theorem for characterizing counterexamples to a declassification policy.

Example 7.2. Consider the following program with h's parity declassified, $h \in \mathsf{Nats}$. We can compute Wlp_P w.r.t. $l = a \in \mathbb{Z}$, and $H_a \overset{\text{def}}{=} \{\langle h, l\rangle \mid h \in \mathbb{V}^{\mathtt{H}}, l = a\}$.

$$P = \left[\begin{array}{lcr} p_0 \rightarrow & \{(h = 0,\ l = a)\ \vee\ (h > 0,\ a = 0)\} & \leftarrow o_0 \\ & \textbf{while } (h > 0) \textbf{ do } (h := h - 1; l := h) \textbf{ endw} & \\ & \{l = a\} & \leftarrow o_1 \end{array}\right.$$

$$Wlp : \left\{\begin{array}{lll} H_0 & \mapsto & \{\langle h, l\rangle \mid h > 0, l \in \mathbb{V}^{\mathtt{L}}\} \cup \{\langle 0, 0\rangle\} = (\mathbb{V}^{\mathtt{H}} \times \{0\}) \cup \{\langle h, l\rangle \mid h > 0, l \neq 0\} \\ H_a & \mapsto & \{\langle 0, a\rangle\} \qquad (a \neq 0) \end{array}\right.$$

Hence, the unstable elements are $\langle even, a\rangle$, $a \neq 0$. In fact, $\langle even, a\rangle \cap Wlp(H_a) = \{\langle 0, a\rangle\} \neq \varnothing$ and $\langle even, a\rangle \not\subseteq Wlp(H_a)$ since, for instance, $\langle 2, a\rangle \notin Wlp(H_a)$. Thus, all the input states where $l = 0$ are not counterexamples to the declassification policy. On the contrary, for any two runs agreeing on input $l \neq 0$, whenever $h_1 = 0$ and $h_2 \in even \smallsetminus \{0\}$, we observe different outputs. Hence, we can distinguish more than the declassified partition $\{even, odd\}$.

The following example (Li and Zdancewic 2005) shows that this approach provides a weakening of noninterference which corresponds to relaxed noninterference. Both approaches provide a method for characterizing the information that flows and that have to be declassified, indeed they both give the same result since they are driven by (parametric on) the particular output observation.

Example 7.3. Consider the program P (Li and Zdancewic 2005) with $sec, x, y : \mathtt{H}$, and $in, out, z : \mathtt{L}$, where $hash$ is a function:

$$P \stackrel{\text{def}}{=} \left[\begin{array}{l} x := hash(sec);\, y := x \ mod\ 2^{64}; \\ \textbf{if } y = in \textbf{ then } out := 1 \textbf{ else } out := 0; \\ z := x \ mod\ 3; \end{array}\right.$$

Suppose that the declassification policy allows to know, about the secret sec only the information about the equality or not between in and $hash(sec)$ mod 2^{64}. Consider the wlp semantics where out, in and z are the public variable and are respectively $a, b, c \in \mathbb{Z}$:

$$P = \left[\begin{array}{lll} p_0 \rightarrow & \{((a = 1, out = a, hash(sec)\ mod\ 2^{64} = b) \vee & \leftarrow o_0 \\ & \quad (a = 0, out = a, hash(sec)\ mod\ 2^{64} \neq b)), & \\ & \quad [hash(sec)\ mod\ 3 = c]\} & \\ & x := hash(sec);\, y := x \ mod\ 2^{64}; & \\ & \{(a = 1, out = a, y = b, [x\ mod\ 3 = c]) \vee & \\ & \quad (a = 0, out = a, y \neq b, [x\ mod\ 3 = c])\} & \\ & \textbf{if } y = in \textbf{ then } out := 1 \textbf{ else } out := 0; & \\ & \{out = a, in = b, [x\ mod\ 3 = c]\} & \\ & z := x \ mod\ 3; & \\ & \{out = a, in = b, [z = c]\} & \leftarrow o_3 \end{array}\right.$$

Let us consider first P without the last assignment to z and consider the input domain formed by the sets $W_{in,1} \stackrel{\text{def}}{=} \{\langle sec, in, out, x, y\rangle \mid in = hash(sec)\ mod\ 2^{64}, out = 1\}$ and $W_{in,0} \stackrel{\text{def}}{=} \{\langle sec, in, out, x, y\rangle \mid in \neq hash(sec)\ mod\ 2^{64}, out = 0\}$. The set of all these domains embodies the declassification policy since it collects together all the tuples such that sec has the same value for $hash(sec)$ mod 2^{64}. Note that the Wlp_P semantics does the following associations: Recall that $H_{in,out} \stackrel{\text{def}}{=} \left\{ \langle sec, in, out, x, y\rangle \,\middle|\, sec, x, y \in \mathbb{V}^{\mathtt{H}} \right\}$

$$Wlp : \begin{cases} H_{in,1} & \mapsto \quad W_{in,1} \text{ if } in = hash(sec)\ mod\ 2^{64} \\ H_{in,0} & \mapsto \quad W_{in,0} \text{ if } in \neq hash(sec)\ mod\ 2^{64} \\ H_{in,out} & \mapsto \quad \varnothing \text{ otherwise} \end{cases}$$

In this case, we can observe that the only information that an attacker can derive about the variable sec is exactly the information allowed to flow by the declassification policy, namely if in is equal or not to $hash(sec)$ mod 2^{64}.

Let us consider, now, also the last assignment, then we have one more variable and we redefine $W_{in,a}$ as sets of tuples containing also z but without any condition on z since it is not considered in the declassification policy. In this case, the wlp semantics does the following associations:

$$Wlp : \begin{cases} H_{in,1} & \mapsto \quad W_{in,1} \cap \left\{ \langle sec, in, out, x, y, z\rangle \,\middle|\, hash(sec)\ mod\ 3 = z \right\} \\ & \qquad \text{if } in = hash(sec)\ mod\ 2^{64} \\ H_{in,0} & \mapsto \quad W_{in,0} \cap \left\{ \langle sec, in, out, x, y, z\rangle \,\middle|\, hash(sec)\ mod\ 3 = z \right\} \\ & \qquad \text{if } in \neq hash(sec)\ mod\ 2^{64} \\ H_{in,out} & \mapsto \quad \varnothing \text{ otherwise} \end{cases}$$

The new elements added to the domain have one more condition on the secret variable *sec*, which can distinguish further the secret inputs by observing the public output. This makes the initial declassification policy unsatisfied.

7.1. *Refining confidentiality policies*

The natural use of the method previously described is for a semantic driven *refinement* of confidentiality policies. The idea is to start with a confidentiality policy stating what can be released in terms of abstract domains (or equivalence relations). In the extreme case, the policy could state that nothing about secret information must be released.
It is worth noting that the refinement process is meaningful only for *semantic* declassification policies, namely those not due to explicit syntactic declassification, since we do not intend to transform the code. Consider, for instance, the program fragment in Example 6.9. In this case, the explicit declassification declassifies $h = 0$ while the semantics releases the parity of h, hence the refinement of the original declassification policy is the greatest lower bound of the abstract domains modelling these two policies (the property characterising whether h is even different from 0, is 0, or is odd), and clearly we would have to transform the code in order to change the declassification policy fixed by the declassify statement.

A consequence of Corollary 5.2 is that whenever $\mathcal{H}^{\phi}$ is *not* forward complete for Wlp_P, more information is released than what declassification ϕ permits. Thus the partition induced on the secret domain by ϕ must be *refined* by the completion process. When we deal with partitioning closures, forward completeness w.r.t. f fails when the partition of the output domain of f contains unstable elements (Mastroeni 2008, Theorem 2). This implies, in particular, that the refinement/completion process, refines each element X of the output partition that is unstable w.r.t. Y, with $X \cap f(Y)$ and $X \smallsetminus f(Y)$ (Mastroeni 2008). In our context, where f is Wlp_P, and the output domain of Wlp_P is the input domain of P, we have that each element $\langle \phi(h), l\rangle$ of $\mathcal{H}^{\phi}$, that is unstable w.r.t. H_a, is refined by $\langle \phi(h), l\rangle \cap Wlp_P(H_a)$ and $\langle \phi(h), l\rangle \smallsetminus Wlp_P(Wlp_P(H_a))$. In this way we obtain a new input domain $\mathcal{H}^{\phi'}$, from which we can derive the corresponding protection policy $\phi' \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$.

In order to derive the refined policy, ϕ', we perform the following steps: (a) Consider the domain, $\mathcal{H}^{\phi'}$, obtained by completion from $\mathcal{H}^{\phi}$; (b) for each $Y \in \mathcal{H}^{\phi'}$ compute sets, $\Pi_l(Y)$, that are parametric on a fixed public value $l \in \mathbb{V}^{\mathtt{L}}$, formally: $\Pi_l(Y) \overset{\text{def}}{=} \{h \in \mathbb{V}^{\mathtt{H}} \mid \langle h, l\rangle \in Y\}$; (c) for each l, compute the partition, $\overline{\Pi}_l$, induced on the secret domain as $\overline{\Pi}_l \overset{\text{def}}{=} \bigwedge_{X\in\mathcal{H}^{\phi'}} \Pi_l(X)$; (d) let $\Pi \overset{\text{def}}{=} \bigwedge_{l\in\mathbb{V}^{\mathtt{L}}} \overline{\Pi}_l$. The declassification policy, ϕ', can now be defined as a refinement, $\mathcal{R}(\phi)$ of ϕ, by computing the *partitioning closure corresponding to* π (Hunt and Mastroeni 2005), i.e., the *disjunctive completion*, $\curlyvee$, of the sets forming the partition: $\phi' = \mathcal{R}(\phi) \overset{\text{def}}{=} \curlyvee(\Pi)$.

For instance, in Example 5.4, each output observation $k = b$ induces the partition $\pi_b = \{\{h \mid h \geq b\}, \{h \mid h < b\}\}$, which is the information released by the single observation. If we consider the set of all the possible observations, then we derive $\Pi = \bigwedge_b \Pi_b = id$, namely we have $\phi = id$.

Proposition 7.4. Let ϕ model the information declassified. If $\langle \mathcal{H}^{\phi}, \mathcal{H}\rangle$ is not complete for Wlp_P, namely if $\mathcal{H}^\phi \circ Wlp_P \circ \mathcal{H} \sqsupset Wlp_P \circ \mathcal{H}$, then $\mathcal{R}(\phi) \sqsubset \phi$, i.e., $\mathcal{R}(\phi)$ is a refinement of ϕ, and it is the minimal transformation of ϕ[7].

Proof. By Corollary 5.2, we have that if $\mathcal{H}^\phi \circ Wlp_P \circ \mathcal{H} = Wlp_P \circ \mathcal{H}$ does not hold then there exists at least one $k \in \mathbb{V}^{\mathtt{L}}$ such that $Wlp_P(H_k) \notin \mathcal{H}^\phi$, where $H_k \overset{\text{def}}{=} \left\{ \langle h, k\rangle \,\middle|\, h \in \mathbb{V}^{\mathtt{H}} \right\}$. Clearly this implies that there exist $\langle h, l\rangle \in Wlp_P(H_k)$ and $h' \in \phi(h)$ such that $\langle h', l\rangle \notin Wlp_P(H_k)$, otherwise $Wlp_P(H_k)$ would be in $\mathcal{H}^\phi$. Namely, we have that $\langle \phi(h), l\rangle \cap Wlp_P(H_k) \neq \varnothing$ and $\langle \phi(h), l\rangle \not\subseteq Wlp_P(H_k)$, which means that the completion refines $\langle \phi(h), l\rangle$ with $\langle \phi(h), l\rangle \cap Wlp_P(H_k)$ and $\langle \phi(h), l\rangle \smallsetminus Wlp_P(H_k)$. Since, in the public part we have the identity, this refinement corresponds to a refinement of $\mathbb{V}^{\mathtt{H}}$, namely the element $\phi(h)$ is refined by $\left\{ h' \in \mathbb{V}^{\mathtt{H}} \,\middle|\, \langle h', l\rangle \in Wlp_P(H_k) \right\} \cap \phi(h)$ and $\phi(h) \smallsetminus \left\{ h' \in \mathbb{V}^{\mathtt{H}} \,\middle|\, \langle h', l\rangle \in Wlp_P(H_k) \right\}$, and this happens for each $\phi(h)$ such that $\left\{ h' \in \mathbb{V}^{\mathtt{H}} \,\middle|\, \langle h', l\rangle \in Wlp_P(H_k) \right\} \cap \phi(h) \neq \varnothing$.

Because ϕ is partitioning, we obtain again a partition, and in particular we have

$$\overline{\Pi}_l = \left\{ X \,\middle|\, X \overset{\text{def}}{=} \left\{ h' \in \mathbb{V}^{\mathtt{H}} \,\middle|\, \langle h', l\rangle \in Wlp_P(H_k) \right\} \cap \phi(h) \neq \varnothing \right\}$$

By construction, it is straightforward to note that $\curlyvee\left(\overline{\Pi}_l\right) \sqsubseteq \phi$, and therefore, since by construction we have $\mathcal{R}(\phi) = \curlyvee(\Pi) \sqsubseteq \curlyvee\left(\overline{\Pi}_l\right)$, we obtain the thesis $\mathcal{R}(\phi) \sqsubseteq \phi$.

Finally, the minimality of the transformation derives from the minimality of the completeness transformation (Giacobazzi and Quintarelli 2001, Sec. 4). $\square$

Next example shows how the refinement works also in contexts where we declassify relations between secrets.

Example 7.5. Consider the program P (Sabelfeld and Myers 2004) with $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$ and its *Wlp* semantics

$$P = \left[\begin{array}{lll} p_0 \to & \quad\{h_1 = a\} & \leftarrow o_0 \\ & h_1 := h_1;\ h_2 := h_1;\ \ldots h_n := h_1; & \\ & \quad\{(h_1 + h_2 + \ldots + h_n)/n = a\} & \\ & avg := (h_1 + h_2 + \ldots + h_n)/n & \\ & \quad\{avg = a\} & \leftarrow o_2 \end{array}\right.$$

$$Wlp_P : \left\{ \begin{array}{lll} X & \mapsto & \varnothing \qquad \text{if } \forall a \in \mathbb{V}^{\mathtt{L}}.\ X \neq H_a \\ H_a & \mapsto & \left\{ \langle a, h_2, \ldots, h_n, a\rangle \,\middle|\, \forall i.\ h_i \in \mathbb{Z},\ avg = a \right\} \end{array} \right.$$

where $H_a \overset{\text{def}}{=} \{\langle h_1, \ldots, h_n, avg\rangle \mid h_i \in \mathbb{Z}, (h_1 + h_2 + \ldots + h_n)/n = avg = a \in \mathbb{Z}\}$. Suppose the input declassification policy releases the average of the secret values, i.e, $\phi(\langle h_1, \ldots, h_n\rangle) \overset{\text{def}}{=} \{\langle h'_1, \ldots, h'_n\rangle \mid (h'_1 + \ldots + h'_n)/n = (h_1 + \ldots + h_n)/n\}$; the policy collects together all the possible secret inputs with the same average value. Hence, the average is the only property that this partition of states allows to observe. Clearly, the

[7] In theory, this refinement can also be computed as the intersection between the policy ϕ and the refinement of the undeclassified policy $\top$. Efficiency comparison between these two approaches is left to the implementation phase of our work.

program releases more. Consider $n = 5$, $h_i \in \{1, ..., 8\}$, and $X = H_4$. The partition induced by $\mathcal{H}^{\phi}(X)$ on the states with $avg = 4$ is $\{\langle 5,2,3,6,4\rangle, \langle 7,3,1,5,4\rangle, \dots\}$. But, following the definition above, we have that $Wlp_P(H_4) = \{\langle 4,3,7,2,4\rangle, \langle 4,8,3,1,4\rangle, \dots\}$. Thus, we need to refine the original policy, completing $\mathcal{H}^{\phi}(X)$ w.r.t. Wlp_P: we add elements $Wlp_P(H_a)$ for all $a \in \mathbb{Z}$. In each such element, h_1 has the particular value a, corresponding to the average. Formally, the domain, $\mathcal{H}^{\phi'}(X)$, contains all the sets $\{\langle h_1, h_2, \dots, h_n, avg\rangle \mid h_1 = avg = a, \forall i > 1.\ h_i \in \mathbb{Z}\}$; $\mathcal{H}^{\phi'}(X)$ distinguishes all tuples that differ in the first secret input, where ϕ' is obtained as disjunctive completion of the computed partition and declassifies the value of h_1. This is the closest domain to ϕ since, if we add any other element in the resulting domain, we would distinguish more than what is necessary, i.e., more than the distinction on the value of h_1. Indeed, still abstracting the average of the elements, we could add other sets of tuples with the same average value, but the only ones that we can add (i.e, which are not yet in the domain) must add some new distinctions. For example, if we add sets like $\{\langle 4,6,1,5,4\rangle, \langle 4,6,3,3,4\rangle, \dots\}$, where also h_2 is fixed, then we allow also to distinguish the value of h_2, which is not released by the program.

Also next example shows how the refinement process works , but moreover it allows us to compare our technique with the (Unno et al. 2006) characterization of counterexamples to declassification policies.

Example 7.6. Consider the following program fragment (Unno et al. 2006):

$$P \overset{\text{def}}{=} \left[\begin{array}{l} \textbf{if } b \textbf{ then } x := h;\ \textbf{else } x := 0; \\ \textbf{if } x = 1 \textbf{ then } l := 1 \textbf{ else } l := 0; \end{array} \right.$$

(Unno et al. 2006) apply a type-based approach for finding the following counterexample to the classic NI policy: $b = b' = \mathit{true}, x = x' = l = l' = 0, h = 0, h' = 1$, where, e.g., $h = 0$ and $h' = 1$ denote the values of h in the two runs of the program. With this input constraint the program violates classic NI. We show that with our approach we can characterize the set of all counterexamples to the policy, which clearly includes the one given above[8]. Let us compute the weakest precondition analysis of the program:

$$P = \left[\begin{array}{lll} p_0 \rightarrow & \{(b = \mathit{false},\ l = 0) \vee (b = \mathit{true},\ h = 1,\ l = 1) & \leftarrow o_0 \\ & \quad \vee (b = \mathit{true},\ h = 0,\ l = 0)\} & \\ & \textbf{if } b \textbf{ then } x := h;\ \textbf{else } x := 0; & \\ & \quad \{(x = 1,\ l = 1) \vee (x = 0,\ l = 0)\} & \\ & \textbf{if } x = 1 \textbf{ then } l := 1 \textbf{ else } l := 0; & \\ & \quad \{l, b, x\} & \leftarrow o_2 \end{array} \right.$$

$$Wlp : \begin{cases} H_{\langle 1,\mathit{false},x\rangle} & \mapsto \varnothing \\ H_{\langle 0,\mathit{false},x\rangle} & \mapsto \left\{ \langle h,l,b,x\rangle \,\middle|\, h \in \mathbb{V}^{\mathtt{H}}, x, l \in \mathbb{V}^{\mathtt{L}}, b = \mathit{false} \right\} \\ H_{\langle 1,\mathit{true},x\rangle} & \mapsto \left\{ \langle h,l,b,x\rangle \,\middle|\, h = 1, x, l \in \mathbb{V}^{\mathtt{L}}, b = \mathit{true} \right\} \\ H_{\langle 0,\mathit{true},x\rangle} & \mapsto \left\{ \langle h,l,b,x\rangle \,\middle|\, h = 0, x, l \in \mathbb{V}^{\mathtt{L}}, b = \mathit{true} \right\} \end{cases}$$

[8] We are comparing the counterexamples we can derive and not the power of the two approaches.

Hence, given the input partition blocks $W_{\langle l,b,x\rangle} \stackrel{\text{def}}{=} \{ \langle h,l,b,x\rangle \mid h \in \mathbb{V}^{\mathtt{H}} \}$, we note that $W_{\langle 1,true,x\rangle} \cap Wlp_P(H_{\langle 1,true,x\rangle}) \neq \varnothing$ and $W_{\langle 1,true,x\rangle} \not\subseteq Wlp_P(H_{\langle 1,true,x\rangle})$. Therefore, any two input tuples $\langle h,l,b,x\rangle$ and $\langle h',l',b',x'\rangle$ such that $\langle h,l,b,x\rangle \in W_{\langle 1,true,x\rangle} \cap Wlp_P(H_{\langle 1,true,x\rangle})$, $\langle h',l',b',x'\rangle \in W_{\langle 1,true,x\rangle} \smallsetminus Wlp_P(H_{\langle 1,true,x\rangle})$ and $\langle l,b,x\rangle = \langle l',b',x'\rangle$ are counterexamples to the noninterference policy for the program P. In particular, this means that the counterexamples are those such that $h = 0$, $h' = 1$, $b = b' = true$, $l = l' \in \mathbb{V}^{\mathtt{L}}$ and $x = x' \in \mathbb{V}^{\mathtt{L}}$, which include the counterexample provided in (Unno et al. 2006).

7.2. *Refining* Narrow Noninterference policies

The method described for checking and refining a security policy is parametric on public observations, but one could carry out the same process on *properties*. If some information about the execution context of the program is present then we can restrict (abstract) the possible observations. These restrictions can be modeled as abstract domains, and therefore by means of narrow noninterference policies. In particular, it has been proved (Giacobazzi and Mastroeni 2005) that the more we observe about public information, the less secret information can be kept secret. This means that a security policy, unsafe in a general context, can become safe if we consider a weaker observation of the public output.

Next example shows a simple situation where we apply our technique for checking a declassification policy in presence of attackers that in output can only observe properties of public data, instead of values.

Example 7.7. Consider the following program fragment (a slight variation of the electronic wallet (Sabelfeld and Myers 2004)):

$$P \stackrel{\text{def}}{=} x := 0; \textbf{ if } h \geq k \textbf{ then } (h := h - k; l := l + k; x := 1)$$

where $h, k : \mathtt{H}$, $x, l : \mathtt{L}$. Consider only the observation of a property of x, and in particular of the equality with 0 in output, i.e., $\rho(\langle x,l\rangle) = \langle \rho_x(x), l\rangle$, where $\rho_x = \{\top, 0, \neq 0, \bot\}$, and the identity in input, i.e., $\eta = id$.
Let us define $H_a \stackrel{\text{def}}{=} \{ \langle h,x,k,l\rangle \mid \rho([\![P]\!](\langle h,x,k,l\rangle)^{\mathtt{L}}) = \langle a,l\rangle \}$ (noting that in output x can be only 0 or 1). Then we have

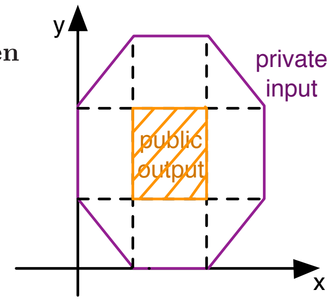
$$Wlp_P : \begin{cases} H_0 & \mapsto \{ \langle h,x,k,l\rangle \mid \rho([\![P]\!](\langle h,x,k,l\rangle)^{\mathtt{L}}) = \langle 0,l\rangle \} \\ & \quad = \{ \langle h,x,k,l\rangle \mid h - k < 0 \} \\ H_{\neq 0} & \mapsto \{ \langle h,x,k,l\rangle \mid \rho([\![P]\!](\langle h,x,k,l\rangle)^{\mathtt{L}}) = \langle \neq 0,l\rangle \} \\ & \quad = \{ \langle h,x,k,l\rangle \mid h - k \geq 0 \} \end{cases}$$

Hence, also in this case we can derive, as we have done before, the input property characterizing the information released about the relation between h and k, which is $\phi = \{\{ \langle h,x,k,l\rangle \mid h-k<0 \}, \{ \langle h,x,k,l\rangle \mid h-k \geq 0 \}\}$.

We can clearly consider more complex abstractions of data. Consider, for instance, the following program P with two secret inputs x, y, and two public outputs x_L, y_L. d, d_x, d_y are constant public inputs with $d_x > d$ and $d_y > d$:

Example 7.8.

$$P \stackrel{\text{def}}{=} \left[\begin{array}{l} \textbf{if}(d \leq x+y \leq d+d_x+d_y \ \wedge\ -d_y \leq x-y \leq d_x) \ \textbf{then} \\ \quad \textbf{if}(x \geq 0 \ \wedge x \leq d) \ \textbf{then} \ x_L := d; \\ \quad \textbf{if}(x > d \ \wedge\ x \leq d_x) \ \textbf{then} \ \ x_L := x; \\ \quad \textbf{if}(x > d_x \ \wedge\ x \leq d_x + d) \ \textbf{then} \ x_L := d_x; \\ \quad \textbf{if}(y \geq 0 \ \wedge y \leq d) \ \textbf{then} \ y_L := d; \\ \quad \textbf{if}(y > d \ \wedge\ y \leq d_y) \ \textbf{then} \ \ y_L := y; \\ \quad \textbf{if}(y > d_y \ \wedge\ y \leq d_y + d) \ \textbf{then} \ y_L := d_y; \end{array}\right.$$



Instead of concrete inputs and outputs, we might want to track *properties*, e.g., in what *interval* a particular variable lies. The figure above represents the input and output of the program in graphical form: the program transforms an input property, namely, an *octagon* (in the secret variables x, y, represented by sets of constraints of the form $\pm x \ \pm y \leq c$) to an output property, namely, a *rectangle* (in the variables x_L, y_L): Thus if we take Wlp_P w.r.t. the *property of intervals* – this corresponds to rectangles in the 2-dimensional space – then the Wlp_P semantics returns an *octagon abstract domain* (Miné 2006), i.e., we derive an octagonal relation between the two secret inputs. Thus the security policy has to declassify at least the octagon domain in order to make the program secure.

Note that, by abstracting the public domain we can make finite the amount of possible observations of the attacker; in practice, this means that when we compute $Wlp(\rho(l) = A)$ we are guaranteed that there will be finitely many Wlp computations whenever the abstract domain is finite.

This example shows that we can combine narrow non interference with declassification in the following completeness equation: Let $\mathcal{H}_\rho \stackrel{\text{def}}{=} \lambda X.\mathbb{V}^{\mathtt{H}} \times \rho(X^{\mathtt{L}})$ (Giacobazzi and Mastroeni 2005) and $\mathcal{H}^\phi_\eta \stackrel{\text{def}}{=} \lambda X.\phi(H_{\eta(l)}) \times \eta(l)$, where $H_{\eta(l)} \stackrel{\text{def}}{=} \{h \mid \eta(l') = \eta(l), \langle h, l'\rangle \in X\}$. Then

$$\mathcal{H}^\phi_\eta \circ Wlp_P \circ \mathcal{H}_\rho = Wlp_P \circ \mathcal{H}_\rho$$

8. An algorithmic approach to declassification refinement

In Sects. 5 and 7 we have seen how we can verify and refine confidentiality policies that admit some leak of secret information. The whole study is done by considering post functions semantics and modelling DNI as a completeness problem. On the other hand, the strong relationship between completeness and stability permits a view of the completeness refinement process for partitions, and hence for declassification, from an *algorithmic* point of view. Indeed, there exists a well known algorithm, the Paige-Tarjan (PT) algorithm (Paige and Tarjan 1987), which allows to find the coarsest stable partition, that is a refinement of an unstable one; in other words it finds the coarsest bisimulation of a given partition. The relation between this algorithm and completion refinement has been extensively studied (Mastroeni 2008; Ranzato and Tapparo 2005; Giacobazzi and

Quintarelli 2001) and has led to the discovery of a strong relation between completeness and strong preservation in *abstract model checking* (Ranzato and Tapparo 2005).

In order to understand how we can exploit these connections for giving an algorithmic approach to the refinement of declassification policies, we note that stability corresponds to $\mathcal{B}$ completeness w.r.t. the post function of a given transition system(Ranzato and Tapparo 2005; Mastroeni 2008). On the other hand, NI and also DNI are characterized as $\mathcal{B}$ completeness problems w.r.t. a (family of) post functions (Theorem 5.1 and Corollary 6.2). This means that, the PT algorithm, which, by inducing stability by partition refinement, induces completeness for partitioning closures, allows also to refine declassification policies for making DNI hold.

The Paige Tarjan algorithm is a well known algorithm for computing the coarsest bisimulation of a given partition and works as follows. Consider a relation $\mathtt{R}$ such that $f = \widetilde{pre}(\mathtt{R})$. The algorithm is provided below, where P is a partition, $\mathtt{PTSplit}_{\mathtt{R}}(S, P)$ partitions each unstable block in P w.r.t. $\mathtt{R}$ with $B \cap f(S)$ and $B \smallsetminus f(S)$, while $\mathtt{PTRefiners}_{\mathtt{R}}(P)$ is the set of all the blocks in P, or obtained as union of blocks in P, which make other blocks unstable. This algorithm has been shown to be a forward completeness problem

$$\mathtt{PTSplit}_{\mathtt{R}}(S,P) : \begin{cases} \text{Partition obtained from } P \text{ by replacing} \\ \text{each block } B \in P \text{ with } B \cap f(S) \text{ and } B \smallsetminus f(S) \end{cases}$$

(with P : Partition)

$$\mathtt{PTRefiners}_{\mathtt{R}}(P) \overset{\text{def}}{=} \left\{ S \;\middle|\; P \neq \mathtt{PTSplit}_{\mathtt{R}}(S,P) \;\wedge\; \exists \{B_i\}_i \subseteq P.\, S = \textstyle\bigcup_i B_i \right\}$$

```
PT-Algorithm_R :
    while ($P$ is not R-stable) do
        choose $S \in \mathtt{PTRefiners}_{\mathtt{R}}(P)$;
        $P := \mathtt{PTSplit}_{\mathtt{R}}(S, P)$;
    endwhile
```

Fig. 4. A generalized version of the PT algorithm.

for the function f (Ranzato and Tapparo 2005), and equivalently a backward completeness problem for the adjoint of f (Giacobazzi and Quintarelli 2001), which is exactly a post function. Moreover, the notion of stability has been generalized to different input and output partitions (Mastroeni 2008), and, as a consequence we can trivially generalize also the algorithm to different partitions, respectively in input and in output (see Sec. 7). The following example, where the $\widetilde{pre}$ is the *Wlp* semantics, shows how the algorithm works in this case.

Example 8.1. Consider Example 7.2. For this example we showed that this declassification policy is not sufficient for insuring noninterference. Now we show that the same result can be obtained by using the PT-algorithm on the partition induced by the declassification policy in input, i.e., $P_I = \{even, odd\}$, and on the output partition induced by the output observation, namely the partition $P_O = \{H_a \mid a \in \mathbb{V}^{\mathtt{L}}\}$. This partition, is not *stable* w.r.t. the wlp semantics and to the output partition, because:

$$even \cap\; Wlp(H_a)_{|_{\mathtt{H}}} \neq \varnothing \;\wedge\; even \not\subseteq Wlp(H_a)_{|_{\mathtt{H}}}$$

where $a \neq 0$ and $Wlp(H_a)_{|_{\mathtt{H}}}$ represents the result of $Wlp(H_a)$ restricted to secret data. In fact, H_a, with $a \neq 0$ belongs to $\texttt{PTRefiners}_{\texttt{R}}(P_I) = \{\ S \in P_O \,\big|\, P_I \neq \texttt{PTSplit}_{\texttt{R}}(S, P_I) \ \wedge\ \exists \{B_i\}_i \subseteq P_O.\, S = \bigcup_i B_i\ \}$, hence P_I is refined substituting the block *even* with the new blocks $\{0\} = \mathit{even} \cap \mathit{Wlp}(H_a)_{|\texttt{H}}$ and $\mathit{even} \smallsetminus \{0\} = \mathit{even} \smallsetminus \mathit{Wlp}(H_a)_{|\texttt{H}}$. At this point the input partition is stable and indeed it is $P_I' = \{\{0\}, \mathit{Even} \smallsetminus \{0\}, \mathit{odd}\}$, which says that, not only can we distinguish even secret inputs from the odd ones, but also we can distinguish between 0 and different from 0 inside the set of even numbers.

Next example shows again the refinement technique, this time on the program fragment proposed by (Zdancewic and Myers 2001), in order to show the relation between the two refinement techniques.

Example 8.2. Consider the following transition system (Zdancewic and Myers 2001) which uses a password system to launder confidential information:

$$\begin{array}{lcll}
\langle t, h, p, q, r\rangle & \mapsto & \langle t, h, p, q, r\rangle & \\
\langle 0, h, p, p, 0\rangle & \mapsto & \langle 1, h, p, p, 1\rangle & \\
\langle 0, h, p, p, 1\rangle & \mapsto & \langle 1, h, p, p, 0\rangle & \\
\langle 0, h, p, q, 0\rangle & \mapsto & \langle 1, h, p, q, 0\rangle & p \neq q \\
\langle 0, h, p, q, 1\rangle & \mapsto & \langle 1, h, p, q, 1\rangle & p \neq q
\end{array}$$

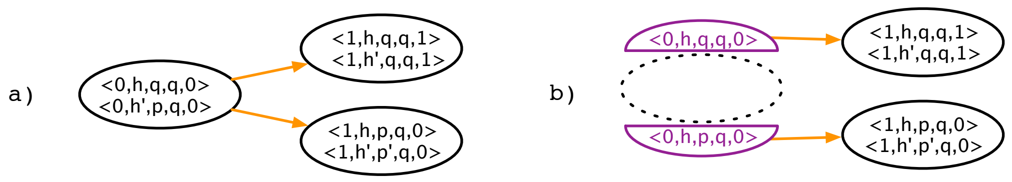
where $t \in \{0, 1\}$ is the time (1 indicates that the program has been executed), $r \in \{0, 1\}$ denotes the result of the test (it is left unchanged if the test of equality between the password p and the query q fails). The public variables are t, q, r, hence the partition induced by $\mathcal{H}$ is:

$$\langle t, h, p, q, r\rangle \ \equiv\ \langle t', h', p', q', r'\rangle \text{ iff } t = t' \ \wedge q = q' \ \wedge\ r = r'$$

The above says we are considering two states that are $\texttt{L}$ indistinguishable (as in ordinary NI). By checking completeness we characterize the information that can be released. For example, consider the set of possible input states which are in the same equivalence class and for which the state $\langle 0, h, p, q, 0\rangle$ is a representative. Applying the transition rules, we see (below, left) that this state reveals a different public output. Thus there is a leakage of confidential information. In order to characterize *what* information is released we complete the domain $\mathcal{H}$ by $\widetilde{\mathrm{pre}}_P$ (below, right):

$$\langle 0, h, p, q, 0\rangle \mapsto \left\{ \begin{array}{l} \langle 1, h, p, p, 1\rangle \\ \langle 1, h, p, q, 0\rangle \end{array}\right. \qquad \widetilde{\mathrm{pre}}_P : \left\{ \begin{array}{ll} \langle 1, h, p, p, 1\rangle \mapsto \langle 0, h, p, p, 0\rangle & \\ \langle 1, h, p, q, 0\rangle \mapsto \langle 0, h, p, q, 0\rangle & p \neq q \end{array}\right.$$

Hence we have to refine the original partition by adding the new blocks $\langle 0, h, p, p, 0\rangle$ and $\langle 0, h, p, q, 0\rangle$ where $p \neq q$, i.e., we release the information whether $p = q$ or $p \neq q$.



In the picture above we have represented the partition of states induce by the input and output observation (in this case they coincide). We can note that this partition is not stable w.r.t. the function $\widetilde{pre}_P$, in particular we have that:

$$\begin{array}{c}\left\{\ \langle 0,h,p,q,0\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}\cap\widetilde{pre}_P(\left\{\ \langle 1,h,q,q,1\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\})\neq\varnothing\ \wedge\\ \left\{\ \langle 0,h,p,q,0\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}\not\subseteq\widetilde{pre}_P(\left\{\ \langle 1,h,q,q,1\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\})\end{array}$$

Hence, the PT-algorithm splits the block $\left\{\ \langle 0,h,p,q,0\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}$ into the two new blocks

$$\begin{array}{rl}\left\{\ 0,h,q,q,0 \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}= & \left\{\ \langle 0,h,p,q,0\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}\cap\\ & \widetilde{pre}_P(\left\{\ \langle 1,h,q,q,1\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\})\\ \left\{\ 0,h,p,q,0 \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}},p\neq q\ \right\}= & \left\{\ \langle 0,h,p,q,0\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\}\smallsetminus\\ & \widetilde{pre}_P(\left\{\ \langle 1,h,q,q,1\rangle \ \middle|\ h,p\in\mathbb{V}^{\mathtt{H}}\ \right\})\end{array}$$

This result corresponds exactly to what we would obtain by considering the completeness refinement. Moreover, this corresponds also to the result that Zdancewic and Myers (Zdancewic and Myers 2001) obtain in their work by refining the input partition induced by $\equiv$.

We can use the relation between the PT-algorithm and the abstract model checking (AMC) refinement of models for strong preservation (Ranzato and Tapparo 2005) as the bridge between noninterference and AMC. AMC techniques are usually applied to Kripke structures. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in the state. The Kripke model for a program corresponds to the standard transition system associated with the program where states are labelled with the values of the variables. The connection between declassification and AMC suggests the use of existing algorithms for AMC in order to derive the information released by a system, whenever the confidential information is fixed. Indeed, the existence of a spurious counterexample in the AMC (abstraction corresponding to the declassification policy) corresponds to the existence of an insecure information flow in the concrete system. Suppose we interpret the initial abstract domain of a system as a declassification policy (the distinction between all the states mapped to different properties is declassified). Then whenever an AMC algorithm finds a spurious counterexample it means that there is a breach in the security, and hence some more secrets, i.e., some more distinctions among states, are released. For instance, in the example above, the given trace (from $\langle 0,h,p,q,0\rangle$) would be identified as a spurious counterexample, and the refinement for erasing it is exactly the refinement we describe. When no more spurious counterexamples exist, then we have characterized, in the resulting abstract domain, the secure declassification policy.

9. Discussion

In this paper we exploit completeness of abstract interpretation for modelling noninterference for confidentiality policies based on declassification. Starting with Joshi and Leino's semantic formulation of NI (Joshi and Leino 2000), it is possible to characterize NI as a problem of $\mathcal{B}$-completeness for denotational semantics (Giacobazzi and Mastroeni 2005). This paper provides an equivalent formulation of NI as $\mathcal{F}$-completeness for the wlp semantics, and extends the formulation to declassification. Semantically, we represent a declassification policy as an abstraction of the `H` inputs that induces a partition on them. A program that satisfies the policy is guaranteed not to expose distinctions within a partition block. $\mathcal{F}$-completeness formalizes "not exposing distinctions". The advantage of our formalization, compared to other approaches, is that we can associate with each possible public observation the exact secret released. Moreover, the strong connection between completeness and declassification, together with the connection between completeness and the Paige-Tarjan algorithm, allows to characterize an algorithmic approach for checking and refining declassification policies. This connection leads also to the possibility of using standard abstract model checking techniques for implementing our approach. In particular, model checking can be applied to generic finite state systems, and abstractions allow to consider even infinite state systems. As future work, we are studying the practical use of these techniques applied to more complex systems.

The relation between the abstract interpretation approach to NI (Giacobazzi and Mastroeni 2004; Giacobazzi and Mastroeni 2005) and many extant approaches for noninterference and declassification has been studied by means of examples (Hunt and Mastroeni 2005; Mastroeni 2005). Sabelfeld and Sands note that most extant proposals suffer from lack of a compelling semantics for declassification. In earlier work they use the PER model (Sabelfeld and Sands 2001) for defining selective dependency (Cohen 1977) by means of equivalence relations instead of abstract domains. They also show, via an example, that the PER model can be used to show that nothing more is learnt by an attacker than what the policy itself releases (Sabelfeld and Sands 2007); in our model we derive this formally (Corollary 5.2) and also show how, in the case where a policy is not satisfied, counterexamples may be generated and the policy may be refined. Joshi and Leino (Joshi and Leino 2000) introduce *abstract variables* in order to obtain a more general notion of security. In this case they substitute the secret variables with functions, i.e., properties, of them. This corresponds to abstract noninterference where we fix what we want to protect instead of what we admit to flow (Giacobazzi and Mastroeni 2004; Mastroeni 2005), hence it is not helpful for computing what information is released. Darvas et al. (Darvas et al. 2005) use dynamic logic to dynamically analyze the declassification property. The information flow property is modelled as a dynamic logic formula. Next, they fix some declassifying preconditions and execute the analysis. If the analysis succeeds then there is an upper bound on the information disclosed; otherwise the precondition must be refined. Because of the connections of completeness to PT, our approach can provide a more systematic method for designing and refining these preconditions. Our approach differs from quantitative characterizations (Clark et al. 2004; Di Pierro et al. 2002) of the information released since we provide a *qualitative* analysis of the leaked secrets.

We have been directly inspired by the work on gradual release (Askarov and Sabelfeld 2007a). In addition to reasoning about the *where* dimension of declassification (as in gradual release), we can reason about the *what* dimension of declassification also. In recent work, building directly on gradual release (Banerjee et al. 2008) show a way to specify the *when*, *what* and *where* policies: *"when"* specifies conditions under which downgrading is allowed. The specification itself is based on a logic for information flow due to (Amtoft et al. 2006). The end-to-end semantic property, like ours, is trace-based and is based on a model that allows observations of low projections of intermediate states as well as termination. Our work does not consider the *when* dimension of declassification. On the other hand, (Banerjee et al. 2008) do not show a way to *compute* the information released by a program or to check whether the information released is more than what is specified by policy.

Unno et al. (Unno et al. 2006) have proposed a method for automatically finding counterexamples of secure information flow, which combines security type-based analysis for standard NI and model checking. Our context is more general, since standard NI is a particular case of DNI. Nevertheless, as future work, we plan to investigate how their approach to NI can be combined with ours for DNI.

Alur et al. (Alur et al. 2006) consider preservation of secrecy under refinement and present a simulation-based technique to show when one system is a refinement of another w.r.t. secrecy. They contend that their approach is flexible because it can express arbitrary secrecy requirements. In particular, if the specification does not maintain secrecy of a property then the implementation does not need to either. Our notion of refinement is slightly different: if a program leaks more information than the policy, we consider how the policy might have to be refined to admit the program. It is possible that there might be strong connections to their work and we plan to explore these connections.

In other future work, we plan to investigate two other applications of our approach. First, we can note that in this paper we talk only of narrow noninterference, while in the original work about the weakening of noninterference by abstract interpretation another notion is introduced: Abstract noninterference (Giacobazzi and Mastroeni 2004). This notion was introduced for two reasons: (a) Narrow noninterference is based on the computation of the *concrete semantics*, which can only be observed by an attacker; (b) Narrow noninterference generate false alarms due to the abstract observation, namely there can be secure situations rejected as insecure. In order to avoid false alarms and to allow to analyze the semantics of the program, abstract noninterference considers attackers that can compute *abstract semantics* of programs, namely attackers that can statically analyze the code of a program. Also for this notion, the connection with completeness is very strong (Giacobazzi and Mastroeni 2005) and we would like to investigate how we can generalize this connection in order to deal with different semantic and protection policies, exactly as we have done in this work for narrow noninterference.

The second direction we would like to follow is the understanding of what happens when we deal with active attackers. An attacker is active when it can both observe and modify the executions of programs. Our idea is that of modelling active attackers as semantic transformers, and whenever we know the program points where the attacker can make its activity (Myers et al. 2004), then we think that *Wlp* is a good semantic model where it is possible to characterize both observational and active power of attackers.

Finally, we plan to automate our approach. This would involve the development of backwards analysis techniques for the particular abstract domains of interest. A hint of this development can be seen in Example 7.8. In the example we need a backwards analysis for the interval abstract domain. In general, however, it is not clear how to compute computable backwards analyses for arbitrary abstract domains. So we might need to restrict the kinds of properties that can be observed. This is an area of future work.

10. Acknowledgement

This is a revised version of a paper that appeared in the Proceedings of the Twenty-Third Conference on Mathematical Foundations of Programming Semantics (MFPS), 2007 (Banerjee et al. 2007). It is a pleasure to acknowledge several stimulating discussions with Roberto Giacobazzi who was a co-author of the MFPS paper. We also thank the anonymous referees for their careful reading of the manuscript and for their suggestions on improving the presentation.

Mastroeni was partially supported by the PRIN projects SOFT.

Banerjee was partially supported by US NSF grants CCF-0296182, ITR-0326577, CNS-0627748; by Microsoft Research, Redmond; by Madrid Regional Government projects S2009TIC-1465 Prometidos, TIN2009-14599-C03-02 Desafios; by EU IST FET Project 231620 Hats.

References

R. Alur, P. Cerny, and S. Zdancewic. Preserving secrecy under refinement. In *Proc. of the Internat. Colloq. on Automata, Languages and Programming* (*ICALP '06*), volume 4052 of *LNCS*, pages 107–118, Berlin, 2006. Springer-Verlag.

T. Amtoft, S. Bandhakavi, and A. Banerjee. A Logic for Information Flow in Object-Oriented Programs. In *Proc. of the 33rd Annual ACM Symposium on Principles of Programming Languages*, pages 91-102. ACM Press.

A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, Los Alamitos, Calif., 2007. IEEE Comp. Soc. Press.

A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and the where dimensions of information release. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM Press.

A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics* (*MFPS '07*), volume 1514 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2007. Elsevier.

A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, 2008.

D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp. Badford, MA, 1973.

D. Clark, S. Hunt, and P. Malacaria. Quantified interference: Information theory and information flow (extended abstract), 2004.

E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5):133–139, 1977.

P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252, New York, 1977. ACM Press.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282, New York, 1979. ACM Press.

A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing: Second International Conference (SPC 2005)*, volume 3450, pages 193–209, Berlin, 2005. Springer-Verlag.

A. Di Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 1–17, Los Alamitos, Calif., 2002. IEEE Comp. Soc. Press.

E.W. Dijkstra. Guarded commands, nondeterminism and formal derivation of programs. *Comm. of The ACM*, 18(8):453–457, 1975.

R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*, pages 186–197, New York, 2004. ACM-Press.

R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In S. Sagiv, editor, *Proc. of the European Symp. on Programming (ESOP '05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310, Berlin, 2005. Springer-Verlag.

R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th Internat. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373, Berlin, 2001. Springer-Verlag.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.

J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, Los Alamitos, Calif., 1984. IEEE Comp. Soc. Press.

G. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dept. of Computer Science, Univ. of Toronto, 1975.

S. Hunt and I. Mastroeni. The PER model of abstract non-interference. In C. Hankin and I. Siveroni, editors, *Proc. of The 12th Internat. Static Analysis Symp. (SAS '05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 171–185, Berlin, 2005. Springer-Verlag.

R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.

G. Kahn. Natural semantics. In *Proc. of Symp. on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Berlin, 1987. Springer-Verlag.

P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. of the 32st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '05)*, pages 158–170, New York, 2005. ACM-Press.

I. Mastroeni. On the rôle of abstract non-interference in language-based security. In K. Yi, editor, *Third Asian Symp. on Programming Languages and Systems (APLAS '05)*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433, Berlin, 2005. Springer-Verlag.

I. Mastroeni. Deriving bisimulations by simplifying partitions. In *Proc. of the 9th Internat. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, volume 4905 of *Lecture Notes in Computer Science*, pages 147–171, New York, 2008. Springer-Verlag.

A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java information flow. Software release.*

A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Symp. on Security and Privacy*, pages 21–34, Los Alamitos, Calif., 2004. IEEE Comp. Soc. Press.

R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):977–982, 1987.

F. Ranzato and F. Tapparo. An abstract interpretation-based refinement algorithm for strong preservation. In N. Halbwachs and L. Zuck, editors, *Proc. of TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 140–156, Berlin, 2005. Springer-Verlag.

A. Sabelfeld and A. C. Myers. A model for delimited information release. In N. Yonezaki K. Futatsugi, F. Mizoguchi, editor, *Proc. of the International Symp. on Software Security (ISSS'03)*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191, Berlin, 2004. Springer-Verlag.

A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.

A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. of Computer Security*, 2007.

D. A. Schmidt. Comparing completeness properties of static analyses and their logics. In N. Kobayashi, editor, *Proc. 2006 Asian Programming Languages and Systems Symposium (APLAS'06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 183–199, Berlin, 2006. Springer-Verlag.

H. Unno, N. Kobayashi, and A. Yonezawa. Combining type-based analysis and model checking for finding counterexamples against non-interference. In *Proc. of the Workshop on Programming languages and analysis for security (PLAS '06)*, pages 17–26, New York, NY, USA, 2006. ACM Press.

G. Winskel. *The formal semantics of programming languages: an introduction.* MIT press, Cambridge, Mass., 1993.

S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 15–23, Los Alamitos, Calif., 2001. IEEE Comp. Soc. Press.