*Math. Struct. in Comp. Science* (2011), *vol.* 21, *pp.* 1253–1299. 
doi:10.1017/S096012951100020X

# Modelling declassification policies using abstract domain completeness

ISABELLA MASTROENI[†] and ANINDYA BANERJEE[‡]

[†]*Università di Verona, Verona, Italy*
*Email:* isabella.mastroeni@univr.it
[‡]*IMDEA Software Institute, Madrid, Spain*
*Email:* anindya.banerjee@imdea.org

*Received 2 June 2008; revised 15 June 2010*

This paper explores a three dimensional characterisation of a declassification-based non-interference policy and its consequences. Two of the dimensions consist of specifying:

(a) the power of the attacker, that is, what public information a program has that an attacker can observe; and
(b) what secret information a program has that 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 any semantics in Cousot's semantics hierarchy.

To check whether a program satisfies a non-interference policy, one can compute an abstract domain that over-approximates the information released by the policy and then check whether program execution can release more information than 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 required 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 and object fields, are annotated with security labels, `H` (high/private/secret) and `L` (low/public/observable), where the labels denote levels in a two point lattice, $\mathtt{L} \leqslant \mathtt{H}$. 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 authorised to

[†] Isabella Mastroeni was partially supported by the PRIN projects SOFT.
[‡] Anindya Banerjee was partially supported by US NSF grants CCF-0296182, ITR-0326577, CNS-0627748; by Microsoft Research, Redmond; by the Madrid Regional Government projects S2009TIC-1465 Prometidos, TIN2009-14599-C03-02 Desafios; and by the EU IST FET Project 231620 Hats.

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 authorised to read only from channels at level L (H and L) and a principal at level H is not authorised to write to a channel at level L. A more formal description of the Bell and La Padula policies is *non-interference* (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 if and only if they agree on the values of the L variables, but not necessarily on the values of H variables. Hence, 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 through other covert channels, such as power consumption or memory exhaustion, but modes of information transmission like this 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 and program logics – see the survey in Sabelfeld and Myers (2003) and references therein.

The above discussion implicitly assumes that given a program that 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 if it cannot look up values of secret input variables directly?

### 1.1. *First contribution*

We address this question by suggesting that an NI policy is characterised by three dimensions:

(a) the observation policy, that is, what the attacker can observe/discover (also called the 'attacker model') (Section 3.2);
(b) what must be protected – or dually, declassified – which we call the 'protection policy' or dually, the 'declassification policy' (Section 3.3);
(c) the choice of concrete semantics for the program (Section 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, through 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 $5 \leqslant h \leqslant 8$. The choice of semantics also places limits on *where* the attacker might be able to make the observations, for example, only at input–output (for denotational semantics) or all intermediate program points (for trace semantics).

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

Finally, (b) specifies the protection policy, namely, what property of the initial secret input must be protected. This is often 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 organisation 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 honoured at the output, rather than at intermediate program points (in contrast to a trace semantics).

### 1.2. *Second contribution*

Our second contribution (Sections 4 and 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 with respect to the policy.

Moreover, we show that when a program does not satisfy a declassification policy (that is, when completeness fails):

(i) counterexamples that expose the failure can be generated (Section 7);
(ii) there is an algorithm that generates the best refinement of the given policy such that the program respects the refined policy (Section 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 (Section 5) to completeness and stability, which have themselves been related through 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 non-interference (Section 8). This relation is shown also by means of examples, where we use the Paige–Tarjan algorithm to reveal information leaks and to refine the corresponding declassification policy.

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

### 1.4. *Road map of the paper*

In Section 3 we discuss how we can describe a non-interference 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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

an attacker can observe (observation policy) and what we aim to protect (protection policy) – see Figure 2. At this point we introduce a technique based on two main ingredients: completeness in abstract interpretation and weakest precondition semantics, whose relation with non-interference is explained in Section 4.

Then, in Section 5, we explain how we can check declassified non-interference policies by using a weakest precondition semantics-based technique whose theoretical foundations are based on abstract interpretation completeness. In particular, we introduce this technique using policies defined in terms of I/O semantics, that is, 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 formalisation of non-interference (specifically, the one proposed in terms of three dimensions in Section 3), we show, in Section 6, how we can easily extend the technique to non-interference 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 to characterise counterexamples to a given declassified non-interference policy (Section 7) and how we can refine a given declassified policy in order to characterise the most restrictive declassified policy satisfied by the program (Section 8). The interesting aspect of our technique is that these last characterisations can be used in the more general formalisation of non-interference since they are independent of the semantics, in fact, the semantics characterises *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

In this section, we consider the semantics of Winskel's simple imperative language Imp (Winskel 1993), introduce relevant terminology and review the 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 $\mathbb{B} = \{true, false\}$;
— Variables $Var$;
— Arithmetic expressions $Aexp$;
— Boolean expressions $Bexp$;
— Commands $Com$.

We assume that the syntactic structure of numbers is given.

We will use the following conventions:

— $m, n$ range over values $\mathbb{V}$;
— $x, y$ range over variables $Var$;
— $a$ ranges over arithmetic expression $Aexp$;

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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

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

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

Fig. 1. Big-step semantics of Imp

— $b$ ranges over boolean expression *Bexp*;
— $c$ ranges over commands *Com*.

We describe the arithmetic and boolean expressions in *Aexp* and *Bexp* as follows:

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

Finally, the syntax for commands is

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

A complete program is a command. Later in the paper 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 $\perp$, denoting the value of uninitialised variables. We use $\mathit{Vars}(P)$ to denote the set of variables of the program $P \in$ Imp. The standard big-step semantics of Imp is given in Figure 1. In the figure, the notation $\langle c, s\rangle \to s'$ means that command $c$ transforms state $s$ to state $s'$, where a state associates a variable with a value. State $s'$ is called the successor of state $s$. The semantics naturally induces a transition relation, $\to$, on a set of states, $\Sigma$, specifying the relation between a state and its possible successors. We say that $\langle \Sigma, \to\rangle$ is a transition system. States are represented as tuples of values indexed by variables. For example, if $|\mathit{Vars}(P)| = n$, then $\Sigma$ is a set of $n$-tuples of values, that is, $\Sigma = \mathbb{V}^n$. We abuse notation by using $\perp$ to denote the state where all the variables are undefined.

*Traces – maximal trace semantics.* In the following, we use $\Sigma^+$ and $\Sigma^\omega \stackrel{\text{def}}{=} \mathbb{N} \longrightarrow \Sigma$, respectively, to denote the non-empty sets of finite and infinite sequences of symbols in $\Sigma$. Given a sequence $\sigma \in \Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$, its length is denoted by $|\sigma| \in \mathbb{N} \cup \{\omega\}$ and its $i$th element is denoted by $\sigma_i$. 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 $\to$, that is, for all $i < |\sigma|$: $\sigma_i \to \sigma_{i+1}$. If $\sigma \in \Sigma^+$, then $\sigma_\dashv$ and $\sigma_\vdash$ denote the final and initial states, respectively, of $\sigma$. The *maximal trace semantics* (Cousot 2002) of a transition system associated with a program $P$ is $\langle\!|P|\!\rangle \stackrel{\text{def}}{=} \langle\!|P|\!\rangle^+ \cup \langle\!|P|\!\rangle^\omega$, where $\Sigma_\dashv \subseteq \Sigma$ is a set of

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

final/blocking states and $\Sigma_{\vdash}$ denotes the set of initial states for $P$. Hence

$$\langle\!|P|\!\rangle^{\omega} = \{\sigma \in \Sigma^{\omega} |\ \forall i \in \mathbb{N}\,.\,\sigma_i \to \sigma_{i+1}\}$$
$$\langle\!|P|\!\rangle^{+} = \{\sigma \in \Sigma^{+} \mid\ \sigma_{\dashv} \in \Sigma_{\dashv},\ \forall i \in [1, |\sigma|).\,\sigma_{i-1} \to \sigma_i\}.$$

The denotational semantics is obtained by abstracting trace semantics to the input and output states only, that is,

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

In this case we assume 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 $\Phi$, the weakest set of pre-conditions $\Phi'$ leading to $\Phi$ after the execution of $P$. In general, the post- and pre-conditions are sets of states, hence $\mathit{Wlp}(P, \Phi) = \Phi'$ means that $\Phi'$ is the greatest set of states leading to the states $\Phi$ 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, that is, $\forall x \in C.\, x \leqslant_C \rho(x)$. The set of all upper closure operators on $C$ is denoted by $\mathit{uco}(C)$.

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

The completeness of abstract interpretation based static analysis has its origins in the work of P. and R. Cousot, for example, Cousot and Cousot (1977; 1979), and means that the analysis is as expressive as possible. The following example is taken from Schmidt's excellent survey on completeness (Schmidt 2006). 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 it can express properties like $\mathit{isNonNegative}$ and $\mathit{isNonPositive}$, 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 are two notions of completeness, *backward* ($\mathscr{B}$) and *forward* ($\mathscr{F}$) completeness, according to whether the concrete and abstract computations are compared in the abstract or 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 $\rho$ is a sound abstraction for $f$ provided

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

$\rho \circ f \circ \rho \sqsupseteq \rho \circ f$. For instance, in Example 2.1,

$$\mathscr{H}([\![P]\!](\mathscr{H}(X))) \supseteq \mathscr{H}([\![P]\!](X)),$$

so $\mathscr{H}$ is a sound abstraction for $[\![P]\!]$. Completeness is obtained by demanding equality:

— $\rho$ is a $\mathscr{B}$-complete abstraction for $f$ if and only if $\rho \circ f = \rho \circ f \circ \rho$.
— $\rho$ is a $\mathscr{F}$-complete abstraction for $f$ if and only if $f \circ \rho = \rho \circ f \circ \rho$.

Completeness can be generalised to pairs $(\rho, \eta)$ of abstract domains: $\mathscr{B}$-completeness holds for $(\rho, \eta)$ when $\rho \circ f \circ \eta = \rho \circ f$; $\mathscr{F}$-completeness holds for $(\rho, \eta)$ 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) and Mastroeni (2008) and Schmidt's survey, Schmidt (2006), for details. Basically, $\mathscr{F}$-completeness is obtained by adding all the direct images of $f$ to the output abstract domain; $\mathscr{B}$-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 characterisation of classic NI, Giacobazzi and Mastroeni (2005) noted that classic NI is a completeness problem in abstract interpretation. Joshi and Leino (2000) uses 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 if and only if

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

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 that 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 `H` value is associated with $\top$, that is, the set of all possible `H` values'. We will now explain these ideas using an example.

Let $\mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}$ be the sets of possible `H` and `L` values. In particular, if

$$n = |\{x \in \mathit{Vars}(P) \mid x \text{ is } \mathtt{H}\}|,$$

then $\mathbb{V}^{\mathtt{H}} \stackrel{\text{def}}{=} \mathbb{V}^n$, and analogously for `L` variables. The set of program states is $\Sigma = \mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}}$. $\Sigma$ is implicitly indexed by the `H` variables followed by the `L` variables. For any $X \subseteq \Sigma$, $X^{\mathtt{H}}$ (respectively, $X^{\mathtt{L}}$) is the projection of the `H` (respectively, `L`) variables. `L` indistinguishability

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

of states $s_1, s_2 \in \Sigma$, written $s_1 =_{\mathtt{L}} s_2$, denotes the fact 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* $\mathscr{H}$ on sets of concrete program states $\Sigma$; that is, $\mathscr{H} : \wp(\Sigma) \to \wp(\Sigma)$, where $\wp(\Sigma)$ is ordered by subset inclusion $\subseteq$. For each possible value of an $\mathtt{L}$ variable, $\mathscr{H}$ associates *all* possible values of the $\mathtt{H}$ variables in $P$. Thus $\mathscr{H}(X) = \mathbb{V}^{\mathtt{H}} \times X^{\mathtt{L}}$, 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) as follows, where $[\![P]\!]$ is the concrete, denotational semantics of $P$:

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

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

**Example 2.1.** Let $h_1, h_2 \in \{0,1\}$ and $l \in \{0,1\}$. So $\mathbb{V}^{\mathtt{H}} = \{0,1\} \times \{0,1\}$, $\mathbb{V}^{\mathtt{L}} = \{0,1\}$. Consider any $X \subseteq \Sigma$: for example, let $X = \{\langle 0,0,1\rangle\}$, that is, $X$ denotes the state where $h_1 = 0$, $h_2 = 0$, $l = 1$. So $\mathscr{H}(X) = \mathbb{V}^{\mathtt{H}} \times \{1\}$. Let $P$ be the obviously insecure program, $l := h_1$, so $[\![P]\!](X) = \{\langle 0,0,0\rangle\}$ and $\mathscr{H}([\![P]\!](X)) = \mathbb{V}^{\mathtt{H}} \times \{0\}$. On the other hand,

$$[\![P]\!](\mathscr{H}(X)) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \langle 1,0,1\rangle, \langle 1,1,1\rangle\},$$

so we have $\mathscr{H}([\![P]\!](\mathscr{H}(X))) = \mathbb{V}^{\mathtt{H}} \times \{0,1\}$, and thus $\mathscr{H}([\![P]\!](\mathscr{H}(X))) \supseteq \mathscr{H}([\![P]\!](X))$. Because $\mathscr{H}([\![P]\!](\mathscr{H}(X)))$ contains triples $\langle 1,0,1\rangle$ and $\langle 1,1,1\rangle$ that are not present in $\mathscr{H}([\![P]\!](X))$, the dependence of $l$ on $h_1$ has been exposed. In other words $\mathscr{H}[\![P]\!]\mathscr{H}$ collects in output all the possible observations due to the variation of $h_1$, while $\mathscr{H}[\![P]\!]$ provides the observation for a particular value $h_1$. Hence, if these two sets are different, 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 $\mathscr{H}([\![P]\!](\mathscr{H}(X)))$, two *distinct* values, 0 and 1, of $l$ may be associated.

In this paper, we consider more flexible abstractions than Joshi and Leino considered, 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 non-interference

We mentioned the following three dimensions of NI in the introduction:

(a) the semantic policy;
(b) the observation policy; and
(c) the protection/declassification policy.

These dimensions are represented pictorially in Figure 2. In general, to describe any NI policy for a program, we first fix its semantic policy. The semantic policy consists of the

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

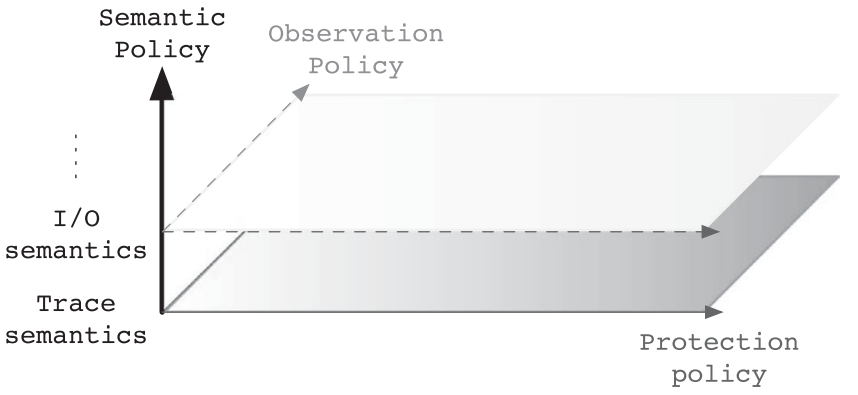



Fig. 2. The three dimensions of non-interference

*concrete semantics* of the program (the solid arrow in Figure 2 shows Cousot's hierarchy of semantics), 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 and protection policies, which 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 $P$[†]):

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

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 the $P$ 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 the program inputs that need to be protected. The observation policy is the identity on the public input–output because the attacker can only 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 can be declassified.

[†] If $P$ is not deterministic, the definition still works if we simply interpret $\llbracket P \rrbracket(s)$ as the set of all the possible outputs starting from $s$.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

### 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. To this end, it is natural to move to a trace semantics. Our first step in this direction is to consider a function *post* as defined below.

Let $\rightarrow_P$ be the transition relation induced by the big-step semantics (Figure 1) of a program $P$. We define

$$post_P \stackrel{\text{def}}{=} \{\langle s,t\rangle \mid s,t \in \Sigma,\ s \rightarrow_P t\}.$$

We can recover classic NI from this definition of $post_P$ as follows. We first define the input–output relation below, which forms a function as $P$ is deterministic†. Let $\rightarrow_P^i$ be the transitive composition of $\rightarrow_P$, $i$ times. This implies, by construction, that $\rightarrow_P^i$ corresponds to the result in the $i$th program point.

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

In other words, $post_P^+$ associates with each initial state $s$ the final state $t$ reachable from $s$. It is then straightforward to note that an equivalent characterisation 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.

*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:

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

where we have used $\langle\!|P|\!\rangle$ to denote the trace semantics of $P$. 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 sequences of states, which are either both finite or both infinite sequences, in which the corresponding states in each sequence are $\mathtt{L}$ indistinguishable. Equivalently, we can use a set of *post* relations: that is, for $i \in \mathsf{Nats}$, we define the family of relations

$$post_P^i \stackrel{\text{def}}{=} \left\{\langle s,t\rangle \;\middle|\; t \in \Sigma,\ s \in \Sigma_\vdash,\ s \rightarrow_P^i t\right\}$$

† If $P$ is not deterministic, 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. Note, moreover, that these definitions hold trivially even if the program is not terminating.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

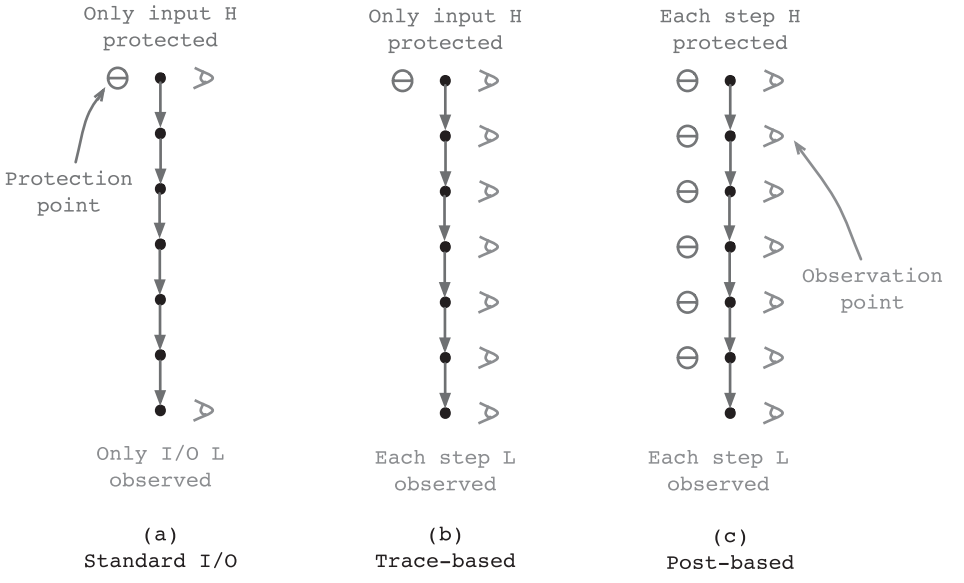



Fig. 3. Different notions of non-interference for trace semantics

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

**Proposition 3.1.** NI on traces holds if and only if for each program point $i$ of program $P$, we have

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

*Proof.* We just need to note that $\langle\!|P|\!\rangle(s_1)^{\mathrm{L}} = \langle\!|P|\!\rangle(s_2)^{\mathrm{L}}$ holds if and only if $post^i_P(s_1) =_{\mathrm{L}} post^i_P(s_2)$ holds for each $i$. □

The *post* characterisation of trace non-interference 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 generalise the semantic policy following this schema.

*General semantic policies.* We have just shown how we can define for denotational and trace semantics a corresponding set of *post* relations fixing protection and observation points. In order to understand how we can generalise this definition, we consider the graphical representation of the situations considered in Figure 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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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 non-interference policy introduced in robust declassification (Zdancewic and Myers 2001), up to an abstraction of traces. This is the notion characterised 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 non-interference for a program, we have to check non-interference 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 at any program point)[†]. Note that (iii) differs form (ii) only in interactive contexts, where there are intermediate private inputs to protect.

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

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

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, with both sets of indexes such that 0 corresponds to the initial program point. In the following, for each index $i \in \mathtt{P}$, we use $p_i$ to denote the corresponding protection point, while for $i \in \mathtt{O}$, we use $o_i$ to denote the corresponding observation point. We can now define a notion of NI parametric on these two sets of program points as follows. Consider any $i$ in $\mathtt{P}$ and any $j$ in $\mathtt{O}$ with $i < j$, and consider

[†] This notion is also the one proposed for approximating abstract non-interference (Giacobazzi and Mastroeni 2004).

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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,

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

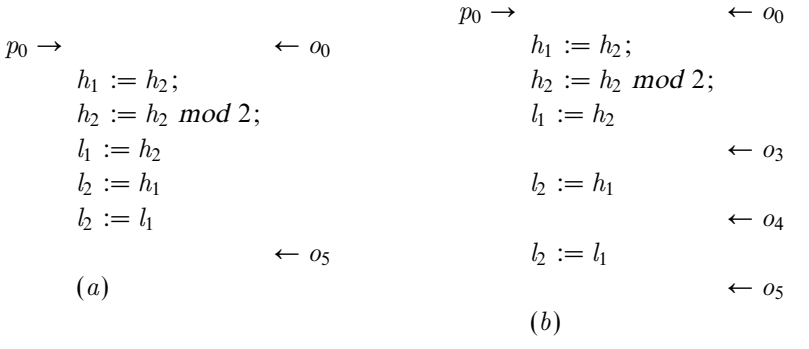
In this way we can also characterise the *where* dimension of declassification, which allows us to specify where we want to protect secret data, for example, in the input or in any possible intermediate state, and where the attacker can observe, for example, in the input–output or in any intermediate state, thus revealing the states in which information leakage happens. However, as we emphasised earlier, 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 rest of the paper. Hence, in the definition above, $\mathtt{P}$ will always be $\{0\}$, namely, we will consider that the initial input states are the only ones to be protected. 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, and that any of the other NI notions described in Section 3 can be formulated as an instance of it, depending on $\mathtt{O}$. In particular, we obtain standard NI by fixing $\mathtt{O} = \{p \mid p \text{ is the final program point}\}$, we obtain the most concrete trace-based NI policy by fixing $\mathtt{O} = \{p \mid p \text{ is any program point}\}$. But we can also obtain intermediate notions depending on $\mathtt{O}$: for example, we obtain gradual release by fixing $\mathtt{O} = \{p \mid p \text{ is a program point corresponding to a low event}\}$. 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 with any more information than observing only the low event states. Nevertheless, the parametric notion allows us to model situations where the environment can, for example, avoid some low event being observed.

**Example 3.3.** Consider the program fragment $P$ with the semantic policies represented in the following picture.




https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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 whether $post^5(s_1)^{\mathtt{L}} = post^5(s_2)^{\mathtt{L}}$, 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 $s_2 = \langle h_1, h_2 = 3, l_1, l_2\rangle$, we have different $l_2$ values in output: *viz.* $l_2 = 0$ and $l_2 = 1$, respectively.

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 whether $post^4(s_1)^{\mathtt{L}} = post^4(s_2)^{\mathtt{L}}$ and $post^3(s_1)^{\mathtt{L}} = post^3(s_2)^{\mathtt{L}}$, and all these tests fail 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* $\rho$ of the public output, for example, parity, and a particular property $\eta$ of the public input, for example, signs. (Technically, $\eta, \rho$ are both closure operators.) Then we obtain a weakening of classic NI as follows:

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

This weakening was first advanced in Giacobazzi and Mastroeni (2004), where it was called *narrow (abstract) non-interference (NNI)*. Classic NI is recovered by setting $\rho$ and $\eta$ 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. In the absence of the 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 will now give an example of how to combine a semantic policy consisting of a trace-based semantics with an observation policy. In other words, we will 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 the 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

$$\langle 3, odd\rangle \to \langle 2, even\rangle \to \langle 1, odd\rangle \to \langle 0, even\rangle \to \langle 0, even\rangle.$$

We can formulate the abstract notion of non-interference on traces by saying that all the execution traces of a program starting from states with different confidential inputs and

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

the same property (say $\eta$) of public input have to provide the same property (say $\rho$) of public parts of reachable states. Therefore, the general notion of narrow non-interference simply consists of abstracting each state of the computational trace. This leads us to the following definition.

**Definition 3.4 (Trace-based NNI).** Given a set of observation points $\mathtt{O}$,

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

The following example shows the meaning of the observation policy, though it will not be considered any further in this paper.

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

It is worth noting that the output abstraction decides what is observable. In the example above, and in the abstract non-interference framework in general, the abstraction $\rho$ is a property of public data, but it can also be interpreted as a further state projection on *some* public variables only. 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 at the observed point: in the example, we could have assumed that only $l_2$ was observable at $o_4$. However, in the following we will assume that the attacker can always observe the whole public state.

### 3.3. *The protection/declassification policy*

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

For a denotational semantics, classic NI says that nothing can be declassified. Formally, just as in the example in Section 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 $\phi$ – the declassifier, which is a function that maps any secret input to its corresponding

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

parity[†]. Then a program $P$ satisfies declassified non-interference (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, for example, in 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 called 'declassification by allowing'. The generalisation of DNI to Definition 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 Definition 3.2. In general, we can consider a different declassification policy $\phi_j$ 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 at $o_j$. Then

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

As we 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 $\phi$ that holds for all the observation points, then $\forall j \in \mathtt{O}.\phi_j = \phi$. On the other hand, with explicit declassification, we have two different choices. We can combine *where* and *what* and assume that the attacker's knowledge can only increase. In this case, for any $i$ in $\mathtt{O}$, using $\phi'_i$ to denote the information declassified in the corresponding program point $i$, the family of declassification policies $\phi_j$ used in Definition 3.6 is $\phi_j = \sqcap_{i \leqslant j} \phi'_i$. In other words, at each observation point, we declassify not only what is explicitly declassified at that point, but also what was declassified previously. For instance, consider the following program fragment:

$$P = \left[\begin{array}{ll} p_0 \rightarrow & \leftarrow o_0 \\ \quad l_1 := \mathit{declassify}(h \geqslant 0); & \\ & \leftarrow o_1 \\ \quad l_2 := \mathit{declassify}(h \leqslant 0); & \\ & \leftarrow o_2 \end{array}\right.$$

Then

$$\phi'_1 = \phi_1 = \{\top, h \geqslant 0, \varnothing\},$$

but at $o_2$ we have

$$\phi'_2 = \{\top, h \leqslant 0, \varnothing\}$$

and

$$\phi_2 = \phi'_1 \sqcap \phi'_2 = \{\top, h \geqslant 0, h \leqslant 0, h = 0, \varnothing\}.$$

So at program point $o_2$ we only explicitly declassify $h \leqslant 0$, but taken together with what was previously declassified, we also declassify $h = 0$. We call this kind of declassification

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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

*incremental* declassification. On the other hand, we can combine *where* and *what* in a stricter way: namely, the information is only declassified at the specific observation point where it is explicitly declared, and not at any following points. In this case it is sufficient to consider as $\phi_j$ exactly the property corresponding to the explicit declassification – we will call this *localised* declassification. This kind of declassification can be useful when we consider the case where the information obtained at the different observation points cannot be combined, for example, if different points are observed by different attackers, as can happen in the security protocol context.

**Example 3.7.** Consider the program fragment in Example 3.3, but with the third line substituted by $l_1 := \mathit{declassify}(h_2)$. Consider the I/O semantic policy in picture ($a$), so we do not consider where the declassification is declared, as 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, $\phi_5(h_1, h_2) = \langle \top(h_1), \mathit{id}(h_2) \rangle \stackrel{\text{def}}{=} \mathit{id}_{h_2}$, that is, we declassify the value of $h_2$. With this declassification policy, non-interference is satisfied. Now consider the semantic policy in picture ($b$). In this case we have two possibilities. If we consider *incremental* declassification, 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, we have $\phi_3 = \mathit{id}_{h_2}$, but also $\phi_4 = \phi_5 = \top$ since at $o_4$ and $o_5$ there are no declassifications. In this case the program is insecure since, as shown in Example 3.3, we release something about $h_2$ at both $o_4$ and $o_5$, namely, its value and its parity, respectively.

## 4. $\mathscr{F}$-completeness and non-interference

In the previous section we characterised non-interference in terms of three dimensions:

— the semantic policy, which depends on the set of observation points and protection points;
— the observation policy, which depends on the property the attacker can observe in these program points; and
— the protection policy, which depends on the property we want to protect in the private input (which is 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. 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 to derive 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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

the connection between NI and completeness of abstract interpretations (Giacobazzi and Mastroeni 2005), which we will now consider.

The example in the introduction (Section 1.2) showed that, as expected, the program $l := h_1$ violates classic NI by demonstrating the fact that the backwards completeness equation (1) does not hold for the program. Equation (1) gives us a way to check *dynamically* whether a program satisfies a confidentiality policy: indeed, we employ the denotational semantics of a program in the process. Can we carry out this check statically?

We will see presently that static checking involves $\mathscr{F}$-completeness, rather than $\mathscr{B}$-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:

$$\mathscr{H} \circ Wlp_P \circ \mathscr{H} = Wlp_P \circ \mathscr{H}. \tag{4}$$

Equation (4) says that $\mathscr{H}$ is $\mathscr{F}$-complete for $Wlp_P$. In other words, consider the abstraction of a concrete input state $X$ through $\mathscr{H}$, which yields a set of states where the secret information is abstracted to 'any possible value'. The equation asserts that $Wlp_P(\mathscr{H}(X))$ is a fixpoint of $\mathscr{H}$, meaning that $Wlp_P(\mathscr{H}(X))$ yields a set of states where each public output is associated with any possible secret input: a further abstraction of the fixpoint (*cf.*, the left-hand side of equation (4)) yields nothing new. Because no *distinctions between 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 non-interference by means of $\mathscr{B}$- and $\mathscr{F}$-completeness are equivalent.

**Theorem 4.1.** $\mathscr{H} \circ [\![P]\!] \circ \mathscr{H} = \mathscr{H} \circ [\![P]\!]$ if and only if $\mathscr{H} \circ Wlp_P \circ \mathscr{H} = Wlp_P \circ \mathscr{H}$.

*Proof.* By Giacobazzi and Quintarelli (2001, Section 4), we know that if $f$ is additive, it has a right adjoint (written $f^+$), so for any $\rho$, we have

$$\rho \circ f \circ \rho = \rho \circ f$$

if and only if

$$\rho \circ f^+ \circ \rho = f^+ \circ \rho.$$

By Cousot (2002, Section 9), we have $[\![P]\!]^+ = Wlp_P$. Then, choosing $\mathscr{H}$ and $[\![P]\!]$ as $\rho$ and $f$, respectively, we are done. □

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

**Example 4.2.** Consider the following program $P$ with $\mathbb{V}^{\mathtt{H}} = \{0,1\} \times \{0,1\}$ and $\mathbb{V}^{\mathtt{L}} = \{0,1\}$:

$$P \stackrel{\text{def}}{=} \begin{bmatrix} \textbf{if } h_1 \neq h_2 & \textbf{then } l := h_1 + h_2 \\ & \textbf{else } l := h_1 - h_2 + 1; \end{bmatrix}$$

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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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

$$\begin{aligned}\mathscr{H}([\![P]\!](\mathbb{V}^{\mathtt{H}} \times \{l\})) &= \mathscr{H}(\mathbb{V}^{\mathtt{H}} \times \{1\}) \\ &= \mathbb{V}^{\mathtt{H}} \times \{1\} \\ &= \mathscr{H}(\langle h_1, h_2, 1\rangle) \\ &= \mathscr{H}([\![P]\!](\langle h_1, h_2, l\rangle)).\end{aligned}$$

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

$$\begin{aligned}\mathscr{H}(Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{1\})) &= \mathscr{H}(\mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}}) \\ &= \mathbb{V}^{\mathtt{H}} \times \mathbb{V}^{\mathtt{L}} \\ &= Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{1\})\end{aligned}$$

$$\begin{aligned}\mathscr{H}(Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{l \neq 1\})) &= \mathscr{H}(\varnothing) \\ &= \varnothing \\ &= Wlp_P(\mathbb{V}^{\mathtt{H}} \times \{l \neq 1\}).\end{aligned}$$

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 $\mathscr{H}$ abstracts the private inputs to $\top$). Thus NI holds.

Therefore, in the following, we will use the wlp semantics to check non-interference 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 non-interference, and it has been proved (Giacobazzi and Mastroeni 2005) that this input domain characterises the *maximal* amount of information disclosed by the program semantics. Intuitively, this refinement process for an abstract domain $O$ with respect to a function $f : I \longrightarrow O$ adds all the direct images of $f$ to $O$, that is, $f(I)$ (Giacobazzi and Quintarelli 2001). For instance, if we consider Example 2.1, we have $Wlp(\langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, \{1\}\rangle) = \langle \{1\}, \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}\rangle$, which means that the observation of 1 in output is due to the private input $h_1 = 1$, which is clearly different from $\mathscr{H} \circ Wlp(\langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, 1\rangle) = \langle \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{H}}, \mathbb{V}^{\mathtt{L}}\rangle$ (and 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 $\{1\}$ and $\{0\}$, and thus $\{\varnothing\}$ (since it is 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 will revisit the example from Section 1.2, but now assume that the security policy for the program $P \stackrel{\text{def}}{=} l := h_1$ allows declassification of $h_1$. In this case the program *would* be secure. Equation (1) must naturally be modified by 'filtering' $\mathscr{H}$ through a declassifier $\phi : \wp(\mathbb{V}^{\mathtt{H}}) \to \wp(\mathbb{V}^{\mathtt{H}})$ that provides an abstraction of the secret

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

inputs. We will write $\mathscr{H}^{\phi} : \wp(\Sigma) \to \wp(\Sigma)$ for this 'filtered $\mathscr{H}$', where $\mathscr{H}^{\phi}$ will be defined formally in Section 5.1. So we require

$$\mathscr{H} \circ [\![P]\!] \circ \mathscr{H}^{\phi} = \mathscr{H} \circ [\![P]\!]. \tag{5}$$

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

As in Section 1.2, let $X$ be $\{\langle 0,0,1\rangle\}$. We are interested in $\phi$'s behaviour on $\{\langle 0,0\rangle\}$, because $\{\langle 0,0\rangle\}$ specifies the values of $h_1, h_2$ in $X$. We have $\phi(\{\langle 0,0\rangle\}) = \{\langle 0,0\rangle, \langle 0,1\rangle\}$ since $\phi$ is the *identity* on what must be declassified (we are releasing the exact value of $h_1$) but $\phi$ is $\top$ on what must be protected, which explains why both $\langle 0,0\rangle$ and $\langle 0,1\rangle$ appear. Now

$$\mathscr{H}^{\phi}(X) = \phi\{\langle 0,0\rangle\} \times X^{\mathtt{L}} = \{\langle 0,0,1\rangle, \langle 0,1,1\rangle\},$$

so

$$[\![P]\!](\mathscr{H}^{\phi}(X)) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle\}$$

and

$$\mathscr{H}([\![P]\!](\mathscr{H}^{\phi}(X))) = \mathbb{V}^{\mathtt{H}} \times \{0\},$$

which is equal to $\mathscr{H}([\![P]\!](X))$. We can show equation (5) for any $X \subseteq \Sigma$, so $l := h_1$ is secure. Note how $\phi$ 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, $\phi$ allows us to expose distinctions *between blocks* at the public output, for example, between $\langle 0,0\rangle$ and $\langle 1,0\rangle$, while in classic NI, $\phi$'s range is $\top$ and *no* distinctions should be exposed.

### 5.1. *Modelling declassification*

The discussion in Section 4 did not provide any motivation for *why* we might want $Wlp_P$, so we will do that now in the context of declassification. Moreover, declassification is a 'property' of the system input, so it is natural to characterise/verify it with a backward analysis from the outputs towards the inputs, that is, 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 $\phi$ is defined as:

$$\begin{aligned} \phi(\varnothing) &= \varnothing \\ \phi\{\langle 0,0\rangle\} &= \phi\{\langle 0,1\rangle\} \\ &= \phi\{\langle 1,0\rangle\} \\ &= \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle\} \end{aligned}$$

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

(that is, we collect together all the elements with the same declassified property)

$$\phi\{\langle 1,1\rangle\} = \mathbb{V}^{\mathtt{H}}$$
$$\phi(X) = \bigcup_{x\in X}(\phi(\{x\})).$$

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. So does $P$ expose any distinctions it should not expose?

To answer this question, we consider $\mathit{Wlp}_P(l = a)$, where $a$ is some generic output value. But why *Wlp*? This is because we can then statically simulate the kind of analysis an attacker would do to obtain the initial values of (or initial relations among) the secret information. And why does $l = a$? This is because it gives us the *most general Wlp, parametric on the output value* (following Gorelick (1975)). Now we note that $\mathit{Wlp}_P(l = a) = (h_1 + h_2 = a)$, and 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, so the program does not satisfy the policy.

So we now consider a declassified confidentiality policy, and model the declassified information by means of the abstraction $\phi$ of the secret inputs, which collects together all the elements with the same property, declassified by the policy. Let $\mathscr{H}^{\phi} : \wp(\Sigma) \to \wp(\Sigma)$ be the corresponding abstraction function. Let $X \in \wp(\Sigma)$ be a concrete set of states and $X^{\mathtt{L}}$ be the $\mathtt{L}$ slice of $X$, and consider any $l \in X^{\mathtt{L}}$. Now define the set

$$X_l \stackrel{\text{def}}{=} \{h \in \mathbb{V}^{\mathtt{H}} \mid \langle h, l\rangle \in X\};$$

that is, given an $l$, $X_l$ contains all the $\mathtt{H}$ values associated with $l$ in $X$. Then the 'declassified' abstract domain $\mathscr{H}^{\phi}(X)$ corresponding to $X$ is defined by

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

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

$$\mathscr{H}^{\phi} \circ \mathit{Wlp}_P \circ \mathscr{H} = \mathit{Wlp}_P \circ \mathscr{H}, \tag{6}$$

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

$$\mathscr{H}^{\phi}(\mathit{Wlp}_P(\mathscr{H}(X))) = \{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \langle 1,0,0\rangle\} \supset \mathit{Wlp}_P(\mathscr{H}(X)).$$

We are now in a position, using Theorem 5.1 below, to connect $\mathscr{H}^{\phi}$ to DNI: the only caveat is that $\phi$ must *partition* the input abstract domain, that is,

$$\forall x.\, \phi(x) = \{y \mid \phi(x) = \phi(y)\}.$$

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

The intuition behind partitioning is that $\phi$'s image on singletons is all we need for deriving the property of any possible set.

**Theorem 5.1.** Consider a partitioning $\phi$. Then $P$ satisfies non-interference declassified by $\phi$ if and only if $\mathscr{H} \circ [\![P]\!] \circ \mathscr{H}^{\phi} = \mathscr{H} \circ [\![P]\!]$.

*Proof.* We first prove that the abstract domain is complete if the program satisfies declassified non-interference. Note that, by hypothesis, we have

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

Note also that all the functions involved in the equation are additive, and thus their composition is additive too. This means that we can prove the equality simply on the singletons. Moreover, since $\phi$ is partitioning, we have $\phi(x) = \{y \mid \phi(x) = \phi(y)\}$, so for any $z \in \phi(x^{\mathtt{H}})$, we have $\phi(z) = \phi(x^{\mathtt{H}})$, so

$$[\![P]\!](\phi(x^{\mathtt{H}}), x^{\mathtt{L}})^{\mathtt{L}} = \bigcup_{z \in \phi(x^{\mathtt{H}})} [\![P]\!](z, x^{\mathtt{L}})^{\mathtt{L}} = [\![P]\!](x^{\mathtt{H}}, x^{\mathtt{L}})^{\mathtt{L}},$$

from the NI hypothesis. Hence the following equalities hold:

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

For the other direction, 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 non-interference declassified by $\phi$. □

Taken together with a straightforward generalisation of Theorem 4.1 to different input and output abstractions, this leads to the following corollary.

**Corollary 5.2.** Consider a partitioning $\phi$. Then $P$ satisfies non-interference declassified by $\phi$ if and only if $\mathscr{H}^{\phi} \circ \mathit{Wlp}_P \circ \mathscr{H} = \mathit{Wlp}_P \circ \mathscr{H}$, that is, $(\mathscr{H}^{\phi}, \mathscr{H})$ is $\mathscr{F}$-complete for $\mathit{Wlp}_P$.

The equality in the corollary asserts that *nothing more is released* by the *Wlp* than is already released by $\phi$. If $\mathscr{F}$-completeness did not hold, but $(\mathscr{H}^{\phi}, \mathscr{H})$ were merely sound, then $\mathscr{H}^{\phi} \circ \mathit{Wlp}_P \circ \mathscr{H} \sqsupseteq \mathit{Wlp}_P \circ \mathscr{H}$. In this case *Wlp* (that is, the right-hand side) releases *more information* (technically, is more concrete) than what is declassified (that is, the left-hand side). 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 $l, h \in \mathsf{Nats}$:

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

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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

Hence, because *Wlp* relates the observed (public) output to the private (secret) inputs, we can derive from the final observation the exact secret released by that observation as follows:

(a) Compute wlp with respect to each observation obtaining a most general predicate on the input states.
(b) Check whether the states described by the wlp are 'more abstract', that is, do not permit more distinctions of secret inputs than those permitted by the policy, and if this is the case, there is no breach.

The following examples show how we can use weakest precondition semantics to check declassified policies.

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

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

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 $\phi(\{h\}) = \mathsf{Nats} \setminus \{0\}$. Thus $\{\{h \mid h \neq 0\}, \{0\}\}$ is the partition induced by $\phi$ on $\mathbb{V}^{\mathtt{H}}$ and we obtain

$$\mathscr{H}^{\phi} = \{\varnothing, \top\} \cup \{\{h \mid h \neq 0\} \times \mathbb{V}^{\mathtt{L}}\} \cup \{\{0\} \times \mathbb{V}^{\mathtt{L}}\}$$

(where the elements $\{\varnothing, \top\}$ are required to make it an upper closure operator). Let $H_a \stackrel{\text{def}}{=} \mathbb{V}^{\mathtt{H}} \times \{a\}$. Now consider the wlp of the program, $\mathit{Wlp}_P(l = a)$, where $a \in \mathbb{V}^{\mathtt{L}}$:

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

So *Wlp* maps the output set of states $H_0$ to the input states $\{\langle h, l\rangle \mid h \ \mathit{mod}\ 2 = 0, l \in \mathbb{V}^{\mathtt{L}}\}$. But this state is no more abstract than the state $\{h \mid h \neq 0\} \times \mathbb{V}^{\mathtt{L}}$ specified by $\mathscr{H}^{\phi}$: for example, it distinguishes $(8, 1)$ from $(7, 1)$, which is not permitted under the policy. Indeed, consider two runs of $P$ with initial values 8 and 7 of $h$ and 1 for $l$. So $\phi(8) = \phi(7)$, but we get two distinct output values of $l$.

**Example 5.4.** Consider (Sabelfeld and Myers 2004)

$$P \stackrel{\text{def}}{=} \textbf{if}\ (h \geqslant k)\ \textbf{then}\ (h := h - k; l := l + k)\ \textbf{else skip},$$

where $l, k : \mathtt{L}$. Also consider

$$H_{a,b} \stackrel{\text{def}}{=} \{\langle h, l, k\rangle \mid h \in \mathbb{V}^{\mathtt{H}},\ l = a,\ k = b\},$$

and suppose the declassification policy is $\top$, that is, nothing should be released. So,

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

$$\mathit{Wlp}_P : H_{a,b} \mapsto \{\langle h, a - b, b\rangle \mid h \geqslant b\} \cup \{\langle h, a, b\rangle \mid h < b\}.$$

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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

## 6. Declassified NI on traces

In this section, we show that we can use the weakest precondition-based approach described in the previous section to check declassification policies with respect to trace-based DNI (Definition 3.6). Before we explain how our approach works on trace-based non-interference by means of examples, we will provide the formal justification for the technique we use on traces.

Since the denotational semantics is the *post* of a transition system where all traces are two-states long (because they are input–output states), we can immediately obtain a straightforward generalisation of the completeness reformulation that also copes with trace-based NI (Definition 3.2). Theorem 6.1 below also shows the connection between completeness and non-interference for traces.

**Theorem 6.1.** Let $\langle \Sigma, \rightarrow_P \rangle$ be the transition system for a program $P$, and $\mathtt{O}$ be a set of observation points. Then non-interference as in Definition 3.2, that is,

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

holds if and only if $\forall j \in \mathtt{O}.\, \mathscr{H} \circ post^j_P \circ \mathscr{H} = \mathscr{H} \circ post^j_P$.

*Proof.* In the following we will simply use $post^j$ to denote the function $post^j_P$. Consider the property

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

that is,

$$\mathscr{H}(post^j(s_1)) = \mathscr{H}(post^j(s_2))$$

for all $s_1 =_{\mathtt{L}} s_2$. Hence, for all $s_1, s_2$ such that $\mathscr{H}(s_1) = \mathscr{H}(s_2)$, we have

$$\mathscr{H}(post^j(s_1)) = \mathscr{H}(post^j(s_2)).$$

By the additivity of $\mathscr{H}$, this implies that for all $s$ and for all $s' \in \mathscr{H}(s)$,

$$\mathscr{H}(post^j(s')) = \mathscr{H}(post^j(s)).$$

Again by the additivity of $\mathscr{H}$, we obtain

$$\mathscr{H}(post^j(\mathscr{H}(s))) = \mathscr{H}(post^j(s)).$$

For the other direction, if we have completeness, that is, $\mathscr{H}(post^j(\mathscr{H}(s))) = \mathscr{H}(post^j(s))$ we can prove that for all $s_1, s_2$ such that $s_1 =_{\mathtt{L}} s_2$, that is, such that $\mathscr{H}(s_1) = \mathscr{H}(s_2)$, we have $\mathscr{H}(post^j(s_1)) = \mathscr{H}(post^j(s_2))$, so the semantics are the same from the public point of view. Indeed, for all $s$ and for all $s' \in \mathscr{H}(s)$ (that is $\mathscr{H}(s) = \mathscr{H}(s')$) we have by completeness that $\mathscr{H}(post^j(s)) = \mathscr{H}(post^j(\mathscr{H}(s)))$ and that $\mathscr{H}(post^j(\mathscr{H}(s'))) = \mathscr{H}(post^j(s'))$. But clearly, by the hypothesis on $s$ and $s'$, we have $\mathscr{H}(post^j(\mathscr{H}(s))) = \mathscr{H}(post^j(\mathscr{H}(s')))$, so the proof is complete, that is, we have shown $\mathscr{H}(post^j(s)) = \mathscr{H}(post^j(s'))$. □

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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

**Corollary 6.2.** Let $\langle \Sigma, \rightarrow_P \rangle$ be the transition system for a program $P \in \textsc{Imp}$ with $\mathtt{O}$ a set of observation points and $\phi \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$ a partitioning closure. Then, non-interference as in Definition 3.6, that is,

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

holds if and only if $\forall j \in \mathtt{O}.\, \mathscr{H} \circ post^j_P \circ \mathscr{H}^{\phi} = \mathscr{H} \circ post^j_P$.

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

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

**Example 6.3.**

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

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. *Localised 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 characterisation

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

of the relation between the *where* and the *what* dimensions of non-interference policies. The idea is to track (using the wlp computation) the information disclosed at 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 at each observable program point. We consider as the set of observable program points $\mathtt{O}$ the same set we used for gradual release, namely, the program points corresponding to low events. However, in general, our technique allows $\mathtt{O}$ to be any set of program points. When there is more than one observation point, we will use the notation $[\Phi]^O$ to denote the fact that the information described by the assertion $\Phi$ can be derived from the set of observation points in $O$.

**Example 6.4.**

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

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

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

We can now use the information disclosed at each observation point, and characterised by computing the wlp semantics, to check if the corresponding declassification policy is satisfied or not. This is achieved 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 the following example.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

**Example 6.5.** Consider the program in Example 6.4, but at $o_3$ replace the statement $l_1 := h_2$ by $l_1 := \mathit{declassify}(h_2)$. In this case the corresponding declassification policy is $\phi_3 = \mathit{id}_{h_2}$. We can now compare this policy with the private information disclosed at $o_3$, which is $h_2 \ \mathit{mod}\ 2 = a$ ($h_2$'s parity) and conclude that the declassification policy at $o_3$ is satisfied because parity is more abstract than identity. Nevertheless, the program releases the information $h_2 = b$ at program point $o_5$. This means that the security of the programs depends on the kind of localised declassification we consider. For incremental declassification, the program is secure because the release is licensed by the previous declassification since the observation point $o_5$ where we release information corresponds to the declassification policy $\phi_5 = \phi_3 = \mathit{id}_{h_2}$, while for localised declassification, the program is insecure since there is no 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 is explicitly declassified.

**Example 6.6.**

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

This program is not secure since from the conjunction of two declassified conditions we can also deduce information about the exact (symbolic) values of the secrets. That is, the information released is *id*, which is more concrete than both

$$\phi_1 = \{\{\langle h_1, h_2\rangle \mid h_1 + h_2 = n\} \mid n \in \mathbb{N}\}$$
$$\phi_3 = \{\{\langle h_1, h_2\rangle \mid h_1 - h_2 = n\} \mid n \in \mathbb{N}\}.$$

## 6.2. *Localised declassification versus gradual release*

In this section we compare the idea introduced above with *gradual release* (Askarov and Sabelfeld 2007a), which was 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 characterises the *knowledge* that can be deduced from the declassification. Suppose $o_1, o_2$ are two consecutive declassification

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

points. Then gradual release ensures that the knowledge at $o_1$, together with the knowledge of the declassification at $o_1$, remains constant until 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 any $\sigma^l \in L(P,s) \stackrel{\text{def}}{=} \{\sigma^l \mid \sigma^l_0 = s\}^\dagger$ we have the following. Letting $|\sigma^l| = n$ and $d_j$ be the indexes of the states obtained after executing an explicit declassification, we have

$$\forall i,j.\ 1 \leqslant i \leqslant n \wedge i \neq d_j \ \Rightarrow\ K_\downarrow(P, \sigma_{\overline{i-1}}) = K_\downarrow(P, \sigma_{\bar{i}})$$

where $\sigma_{\bar{i}}$ is the prefix of $\sigma$ with length $i$, $K_\downarrow(P,\sigma) \stackrel{\text{def}}{=} \{\tau_0 \mid \tau \text{ trace such that } \sigma^l = \tau^l\}$ and $\tau_0$ denotes the initial state of the trace $\tau$.

Note that by Definition 6.7, $L$ makes a *forward* analysis by computing the observed traces starting from a given initial state, while the knowledge $K$ makes a *backward* analysis by computing all the possible input states that 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 and $K$ provides the information that the attacker can derive from this observation. This approach corresponds exactly to the weakest precondition semantics approach proposed in Section 5. We will now show how gradual release works and how it can be compared with our approach using some examples.

**Example 6.8.**

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

The low trace of the computation is

$$\langle l_1, l_2\rangle \rightarrow \langle h_2 \ \mathit{mod}\ 2, l_2\rangle \rightarrow\ \langle h_2 \ \mathit{mod}\ 2, h_2\rangle.$$

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

Example 6.8 is insecure according to gradual release by 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.

† $\sigma^l$ denotes the projection of the trace $\sigma$ only on the low events (low assignment, declassification and termination).

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

The above argument is the same as 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.**

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

The low trace here is $\langle l_1, l_2\rangle \rightarrow \langle h_1 \ \mathit{mod}\ 2, l_2\rangle$. In this case we are releasing the parity since $h_2$ takes its parity just before the declassification statement, but we are explicitly 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. However, it *does* satisfy gradual release since all the released information is leaked at the declassification point, so we never check it.

In contrast to gradual release, our approach considers the *what* dimension. Thus we can determine whether what is released is *more* than what is declassified, even if the information is *only* released at the declassification point. Using $\phi_2$ to denote the property corresponding to the information declassified in the program, which corresponds to the information of being equal or not to 0, consider

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

The only information that flows comes through a declassification, but the information released is the parity while the declassification is $\phi_2$. Because these properties are incomparable, the program is not secure in our approach. This example again shows how our approach can combine the what and where dimensions.

### 6.3. *Localised delimited release*

Localised 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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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.**

$$P = \begin{bmatrix} p_0 \rightarrow & & \leftarrow o_0 \\ & h' := 0; & \\ & \textbf{if } h \textbf{ then } l := \mathit{declassify}(h') \textbf{ else } l := 0; & \\ & & \leftarrow o_2 \end{bmatrix}$$

Intuitively, it is clear that this program is secure since no information about secrets is released. *Without* the declassify statement, localised delimited release recognises this program as secure (Askarov and Sabelfeld 2007b), but *with* the declassification statement in the if branch, the program is labelled as insecure. This happens because the definition of localised release treats declassification with respect to 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, we obtain the following analysis recognising the program as secure since no information is derived about $h$ from the observation of $l$:

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

In general, the problem with localised 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 for which 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 (with one test for each observation point) 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 characterise, for each observation point, the counterexamples to the corresponding declassification policy for which it is not satisfied.

We have advanced the thesis that non-interference is a completeness problem in abstract interpretation. Ranzato and Tapparo (2005) studied completeness from another point of view and showed the strong connection between completeness and the notion of stability – in particular, when dealing with partitioning closures (Mastroeni 2008). In other words, there is a correspondence between completeness and the absence of *unstable* elements

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

of a closure with respect to 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 $X \cap f(Y) = \emptyset$; otherwise $X$ is said to be *unstable*. It is clear that in our declassification context, where we partition the input by $\mathscr{H}^{\phi}$ and the output by $\mathscr{H}$, we do not have the same partition in input and output. This means that we have to consider a slight generalisation of the stability notion where the constraint on 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 $\mathtt{I}$ and $\mathtt{O}$, respectively, then $\Pi_{\mathtt{O}}$ is stable with respect to $f$ and $Y \in \Pi_{\mathtt{I}}$ if $\forall X \in \Pi_{\mathtt{O}}$ we have $X \subseteq f(Y)$ or $X \cap f(Y) = \emptyset$. The understanding of completeness in terms of stability guarantees that if an abstract domain is not complete, at least two of its elements 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 $\phi$; and the element against which we check stability ($Y$ in the definition) is the particular output observation (for example, $l = a$). In general, the particular output will be the set $L$ of all the public data observed in all the different observable points, that is, $L = \{L_i \mid L_i \text{ observation in } o_i\}$. For instance, in Example 3.3, if the attacker observes I/O, in other words, if $\mathtt{O} = \{6\}$, then $L = \{l_1 = l_2 = a\}$ is the particular output observation against which we check the declassification policy $\phi = \phi_3 = id_{h_2}$. On the other hand, if the attacker can observe all the low events ($\mathtt{O} = \{3, 5, 6\}$), then

$$\begin{aligned} L &= \{L_3, L_5, L_6\} \\ &= \{[l_1 = l_2 = a]^{o_6}, [l_2 = b]^{o_5}, [l_2 = c]^{o_3}\} \end{aligned}$$

(we only consider the relevant observations) is the set of public observations against which we check the corresponding declassification policies $\phi_3$, $\phi_5$ and $\phi_6$.

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

*Proof.* We will use $\phi$ to denote any of the declassification policies $\phi_i$. Suppose $\exists L \subseteq \mathbb{V}^{\mathtt{L}}$ (observation at $o_i$) such that (the input states described by) $Wlp(H_L) \notin \mathscr{H}^{\phi}$. 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 $(\phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\}) \cap Wlp(H_L) \neq \emptyset$ since $x$ is in both, and $\phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\} \nsubseteq Wlp(H_L)$ since we have $\langle h, x^{\mathtt{L}} \rangle \in \phi(x^{\mathtt{H}}) \times \{x^{\mathtt{L}}\}$ and $\langle h, x^{\mathtt{L}} \rangle \notin Wlp(H_L)$. Hence, the abstract domain $\mathscr{H}^{\phi}$ is not *stable*. To find a counterexample, consider

$$h_1 \in \phi(x^{\mathtt{H}}) \setminus \{k \mid \langle k, x^{\mathtt{L}} \rangle \in Wlp(H_L)\}$$

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, for example $h_2$, leads to the observation $L$, while all the elements outside the set, for example $h_1$, cannot lead to $L$. But, both $h_1$ and $h_2$ are in the same element of $\phi$, hence they form a counterexample to the declassification policy $\phi$. □

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

The next example shows how we can use the above theorem to characterise counter-examples to a declassification policy.

**Example 7.2.** Consider the following program with $h$'s parity declassified and $h \in \mathsf{Nats}$:

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

We can compute $Wlp_P$ with respect to $l = a \in \mathbb{Z}$, and $H_a \stackrel{\text{def}}{=} \{\langle h, l\rangle \mid h \in \mathbb{V}^{\mathtt{H}}, l = a\}$.

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

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

$$\langle even, a\rangle \notin Wlp(H_a)$$

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 $h_2 \in even \setminus \{0\}$, 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 non-interference, which corresponds to relaxed non-interference. Both approaches provide a method for characterising the information that flows and has to be declassified; in fact, they both give the same result since they are driven by (are parametric on) the particular output observation.

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

$$P \stackrel{\text{def}}{=} \begin{bmatrix} x := \mathit{hash}(sec) y := x \ \mathit{mod}\ 2^{64}; \\ \textbf{if } y = in \textbf{ then } out := 1 \textbf{ else } out := 0; \\ z := x \ \mathit{mod}\ 3; \end{bmatrix}$$

Suppose the declassification policy says that the only information allowed to be known about the secret $sec$ is the equality or not between $in$ and $\mathit{hash}(sec)\ \mathit{mod}\ 2^{64}$. Consider the wlp semantics where $out, in$ and $z$ are the public variable and are $a, b$ and $c$, respectively,

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

with $a, b, c \in \mathbb{Z}$:

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

We first consider $P$ without the final assignment to $z$ and consider the input domain formed by the sets

$$W_{in,1} \stackrel{\text{def}}{=} \{\langle sec, in, out, x, y\rangle \mid in = \mathit{hash}(sec)\ \mathit{mod}\ 2^{64}, out = 1\}$$

and

$$W_{in,0} \stackrel{\text{def}}{=} \{\langle sec, in, out, x, y\rangle \mid in \neq \mathit{hash}(sec)\ \mathit{mod}\ 2^{64}, out = 0\}.$$

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 $\mathit{hash}(sec)\ \mathit{mod}\ 2^{64}$. Note that the $\mathit{Wlp}_P$ semantics makes the following associations:

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

(recall that $H_{in,out} \stackrel{\text{def}}{=} \{\langle sec, in, out, x, y\rangle \mid sec, x, y \in \mathbb{V}^{\mathrm{H}}\}$). 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 $\mathit{hash}(sec)\ \mathit{mod}\ 2^{64}$.

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

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

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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

### 7.1. *Refining confidentiality policies*

The natural use of the previously described method is to produce 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 arising from 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 and 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 $\mathscr{H}^{\phi}$ is *not* forward complete for $Wlp_P$, more information is released than declassification $\phi$ permits. Thus the partition induced on the secret domain by $\phi$ must be *refined* by the completion process. When we deal with partitioning closures, forward completeness with respect to $f$ 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 with respect to $Y$, 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 $\mathscr{H}^{\phi}$ that is unstable with respect to $H_a$ 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 $\mathscr{H}^{\phi'}$, from which we can derive the corresponding protection policy $\phi' \in uco(\wp(\mathbb{V}^{\mathtt{H}}))$.

In order to derive the refined policy $\phi'$, we perform the following steps:

(a) Consider the domain $\mathscr{H}^{\phi'}$ obtained by completion from $\mathscr{H}^{\phi}$.

(b) For each $Y \in \mathscr{H}^{\phi'}$ compute sets $\Pi_l(Y)$ that are parametric on a fixed public value $l \in \mathbb{V}^{\mathtt{L}}$: formally,

$$\Pi_l(Y) \stackrel{\text{def}}{=} \{h \in \mathbb{V}^{\mathtt{H}} \mid \langle h, l\rangle \in Y\}.$$

(c) For each $l$, compute the partition $\overline{\Pi}_l$ induced on the secret domain as

$$\overline{\Pi}_l \stackrel{\text{def}}{=} \bigwedge_{X \in \mathscr{H}^{\phi'}} \Pi_l(X).$$

(d) Let $\Pi \stackrel{\text{def}}{=} \bigwedge_{l \in \mathbb{V}^{\mathtt{L}}} \overline{\Pi}_l$.

The declassification policy $\phi'$ can now be defined as a refinement $\mathscr{R}(\phi)$ of $\phi$ by computing the *partitioning closure corresponding to* $\pi$ (Hunt and Mastroeni 2005), that is, the *disjunctive completion* $\bigvee$ of the sets forming the partition:

$$\phi' = \mathscr{R}(\phi) \stackrel{\text{def}}{=} \bigvee(\Pi).$$

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

For instance, in Example 5.4, each output observation $k = b$ induces the partition

$$\pi_b = \{\{h \mid h \geqslant b\}, \{h \mid h < b\}\},$$

which is the information released by the single observation. If we consider the set of all the possible observations, we derive $\Pi = \bigwedge_b \Pi_b = id$, in other words, $\phi = id$.

**Proposition 7.4.** Let $\phi$ model the information declassified. If $\langle \mathscr{H}^\phi, \mathscr{H} \rangle$ is not complete for $Wlp_P$, in other words, if $\mathscr{H}^\phi \circ Wlp_P \circ \mathscr{H} \sqsupset Wlp_P \circ \mathscr{H}$, then $\mathscr{R}(\phi) \sqsubset \phi$, that is, $\mathscr{R}(\phi)$ is a refinement of $\phi$, and it is the minimal transformation of $\phi$[†].

*Proof.* By Corollary 5.2, we have that if

$$\mathscr{H}^\phi \circ Wlp_P \circ \mathscr{H} = Wlp_P \circ \mathscr{H}$$

does not hold, there is at least one $k \in \mathbb{V}^{\mathtt{L}}$ such that $Wlp_P(H_k) \notin \mathscr{H}^\phi$, where

$$H_k \stackrel{\text{def}}{=} \left\{ \langle h, k \rangle \;\middle|\; h \in \mathbb{V}^{\mathtt{H}} \right\}.$$

It is clear that 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 $\mathscr{H}^\phi$. In other words, we have

$$\langle \phi(h), l \rangle \cap Wlp_P(H_k) \neq \varnothing$$

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 $\mathbb{V}^{\mathtt{H}}$, in other words, the element $\phi(h)$ is refined by

$$\left\{ h' \in \mathbb{V}^{\mathtt{H}} \mid \langle h', l \rangle \in Wlp_P(H_k) \right\} \cap \phi(h)$$

and

$$\phi(h) \smallsetminus \left\{ h' \in \mathbb{V}^{\mathtt{H}} \mid \langle h', l \rangle \in Wlp_P(H_k) \right\},$$

and this happens for each $\phi(h)$ such that

$$\left\{ h' \in \mathbb{V}^{\mathtt{H}} \mid \langle h', l \rangle \in Wlp_P(H_k) \right\} \cap \phi(h) \neq \varnothing.$$

Because $\phi$ is partitioning, we again obtain a partition, and, in particular, we have

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

[†] In theory, this refinement can also be computed as the intersection of the policy $\phi$ and the refinement of the undeclassified policy $\top$. An efficiency comparison between these two approaches has been left to the implementation phase of our work.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

By construction, it is straightforward to see that $\curlyvee(\overline{\Pi}_l) \sqsubseteq \phi$, and thus, since by construction we have

$$\mathscr{R}(\phi) = \curlyvee(\Pi) \sqsubseteq \curlyvee(\overline{\Pi}_l),$$

we get $\mathscr{R}(\phi) \sqsubseteq \phi$ as required.

Finally, the minimality of the transformation follows from the minimality of the completeness transformation (Giacobazzi and Quintarelli 2001, Section 4). □

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

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

$$P = \begin{bmatrix} p_0 \rightarrow & \{h_1 = a\} & \leftarrow o_0 \\ & h_1 := h_1;\ h_2 := h_1;\ \ldots h_n := h_1; & \\ & \{(h_1 + h_2 + \ldots + h_n)/n = a\} & \\ & avg := (h_1 + h_2 + \ldots + h_n)/n & \\ & \{avg = a\} & \leftarrow o_2 \end{bmatrix}$$

$$Wlp_P : \begin{cases} X \mapsto \varnothing & \text{if } \forall a \in \mathbb{V}^{\mathrm{L}}.\ X \neq H_a \\ H_a \mapsto \{\langle a, h_2, \ldots, h_n, a\rangle \mid \forall i.\ h_i \in \mathbb{Z},\ avg = a\} \end{cases}$$

where

$$H_a \stackrel{\text{def}}{=} \{\langle h_1, \ldots, h_n, avg\rangle \mid h_i \in \mathbb{Z}, (h_1 + h_2 + \ldots + h_n)/n = avg = a \in \mathbb{Z}\}.$$

Suppose the input declassification policy releases the average of the secret values, that is,

$$\phi(\langle h_1, \ldots, h_n\rangle) \stackrel{\text{def}}{=} \{\langle h'_1, \ldots, h'_n\rangle \mid (h'_1 + \ldots + h'_n)/n = (h_1 + \ldots + h_n)/n\}.$$

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 us to observe. Clearly, the program releases more. Consider $n = 5$, $h_i \in \{1, ..., 8\}$ and $X = H_4$. The partition induced by $\mathscr{H}^{\phi}(X)$ on the states with $avg = 4$ is $\{\langle 5, 2, 3, 6, 4\rangle, \langle 7, 3, 1, 5, 4\rangle, \ldots\}$. But, following the definition above, we have $Wlp_P(H_4) = \{\langle 4, 3, 7, 2, 4\rangle, \langle 4, 8, 3, 1, 4\rangle, \ldots\}$. Thus, we need to refine the original policy, completing $\mathscr{H}^{\phi}(X)$ with respect to $Wlp_P$: 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 $\mathscr{H}^{\phi'}(X)$ contains all the sets

$$\{\langle h_1, h_2, \ldots, h_n, avg\rangle \mid h_1 = avg = a, \forall i > 1.\ h_i \in \mathbb{Z}\}.$$

$\mathscr{H}^{\phi'}(X)$ distinguishes all tuples that differ in the first secret input, where $\phi'$ is obtained as the disjunctive completion of the computed partition and declassifies the value of $h_1$. This is the closest domain to $\phi$ since if we add any other element in the resulting domain, we would distinguish more than is necessary, that is, 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 (that is, which

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

are not yet in the domain) must add some new distinctions. For example, if we add sets like $\{\langle 4,6,1,5,4\rangle, \langle 4,6,3,3,4\rangle, \ldots\}$, where $h_2$ is also fixed, we also allow the value of $h_2$ to be distinguished, but this is not released by the program.

The next example also shows how the refinement process works, but it also allows us to compare our technique with Unno *et al.* (2006)'s characterisation of counterexamples to declassification policies.

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

$$P \stackrel{\text{def}}{=} \begin{bmatrix} \textbf{if } b \textbf{ then } x := h; \textbf{ else } x := 0; \\ \textbf{if } x = 1 \textbf{ then } l := 1 \textbf{ else } l := 0; \end{bmatrix}$$

Unno *et al.* (2006) applies a type-based approach to find the following counterexample to the classic NI policy: $b = b' = \mathit{true}, x = x' = l = l' = 0, h = 0, h' = 1$, where, for example, $h = 0$ 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 characterise the set of all counterexamples to the policy, which clearly includes the one given above†. We will now compute the weakest precondition analysis of the program:

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

$$\mathit{Wlp} : \begin{cases} H_{\langle 1,\mathit{false},x\rangle} & \mapsto \emptyset \\ H_{\langle 0,\mathit{false},x\rangle} & \mapsto \{\langle h,l,b,x\rangle \mid h \in \mathbb{V}^{\mathrm{H}}, x, l \in \mathbb{V}^{\mathrm{L}}, b = \mathit{false}\} \\ H_{\langle 1,\mathit{true},x\rangle} & \mapsto \{\langle h,l,b,x\rangle \mid h = 1, x, l \in \mathbb{V}^{\mathrm{L}}, b = \mathit{true}\} \\ H_{\langle 0,\mathit{true},x\rangle} & \mapsto \{\langle h,l,b,x\rangle \mid h = 0, x, l \in \mathbb{V}^{\mathrm{L}}, b = \mathit{true}\} \end{cases}$$

Hence, given the input partition blocks $W_{\langle l,b,x\rangle} \stackrel{\text{def}}{=} \{\langle h,l,b,x\rangle \mid h \in \mathbb{V}^{\mathrm{H}}\}$, we have

$$W_{\langle 1,\mathit{true},x\rangle} \cap \mathit{Wlp}_P(H_{\langle 1,\mathit{true},x\rangle}) \neq \emptyset$$

and

$$W_{\langle 1,\mathit{true},x\rangle} \nsubseteq \mathit{Wlp}_P(H_{\langle 1,\mathit{true},x\rangle}).$$

Therefore, any two input tuples $\langle h,l,b,x\rangle$ and $\langle h',l',b',x'\rangle$ such that

$$\begin{aligned} \langle h,l,b,x\rangle &\in W_{\langle 1,\mathit{true},x\rangle} \cap \mathit{Wlp}_P(H_{\langle 1,\mathit{true},x\rangle}) \\ \langle h',l',b',x'\rangle &\in W_{\langle 1,\mathit{true},x\rangle} \setminus \mathit{Wlp}_P(H_{\langle 1,\mathit{true},x\rangle}) \\ \langle l,b,x\rangle &= \langle l',b',x'\rangle \end{aligned}$$

† We are not comparing the power of the two approaches here, just the counterexamples we can derive using them.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

are counterexamples to the non-interference policy for the program $P$. In particular, this means that the counterexamples are those such that $h = 0$, $h' = 1$, $b = b' = \mathit{true}$, $l = l' \in \mathbb{V}^{\mathtt{L}}$ and $x = x' \in \mathbb{V}^{\mathtt{L}}$, which include the counterexample provided in Unno *et al.* (2006).

### 7.2. *Refining narrow non-interference 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, we can restrict (or abstract) the possible observations. These restrictions can be modelled as abstract domains, and therefore by means of narrow non-interference policies. In particular, it has been proved that the more we observe about public information, the less secret information can be kept secret (Giacobazzi and Mastroeni 2005). This means that a security policy, unsafe in a general context, can become safe if we consider a weaker observation of the public output.

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

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

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

where $h, k : \mathtt{H}$, $x, l : \mathtt{L}$. We now just consider the observation of a property of $x$, and, in particular, of the equality to 0 in the output, that is, $\rho(\langle x, l\rangle) = \langle \rho_x(x), l\rangle$, where $\rho_x = \{\top, 0, \neq 0, \bot\}$, and the identity in input, that is, $\eta = \mathit{id}$.

We define

$$H_a \stackrel{\text{def}}{=} \left\{ \langle h, x, k, l\rangle \;\middle|\; \rho([\![P]\!](\langle h, x, k, l\rangle)^{\mathtt{L}}) = \langle a, l\rangle \right\}$$

(noting that $x$ can only be 0 or 1 in the output). Then we have

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

Hence, in this case we can again derive the input property characterising the information released about the relation between $h$ and $k$, which is

$$\phi = \{\{\langle h, x, k, l\rangle \mid h - k < 0\}, \{\langle h, x, k, l\rangle \mid h - k \geqslant 0\}\}.$$

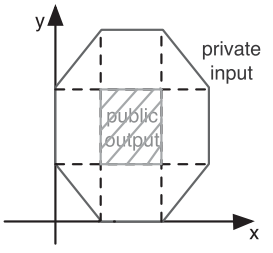
**Example 7.8.** It is clear that we can consider more complex abstractions of data, such as the following program $P$ with two secret inputs $x$ and $y$, and two public outputs $x_L$ and

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

$y_L$, where $d, d_x$ and $d_y$ are constant public inputs with $d_x > d$ and $d_y > d$:

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

Instead of concrete inputs and outputs, we might want to track *properties*, for example, in what *interval* a particular variable lies. The following figure 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 $\pm x \ \pm y \leqslant c$) to an output property, namely, a *rectangle* (in the variables $x_L, y_L$). Thus, if we take $Wlp_P$ with respect to 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), that is, 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 ensure that the number of observations the attacker can make is finite; 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 $\mathscr{H}_\rho \stackrel{\text{def}}{=} \lambda X.\mathbb{V}^{\mathtt{H}} \times \rho(X^{\mathtt{L}})$ (Giacobazzi and Mastroeni 2005) and

$$\mathscr{H}^{\phi}_{\eta} \stackrel{\text{def}}{=} \lambda X.\phi(H_{\eta(l)}) \times \eta(l),$$

where

$$H_{\eta(l)} \stackrel{\text{def}}{=} \{h \mid \eta(l') = \eta(l), \langle h, l' \rangle \in X\}.$$

Then

$$\mathscr{H}^{\phi}_{\eta} \circ Wlp_P \circ \mathscr{H}_\rho = Wlp_P \circ \mathscr{H}_\rho.$$

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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

$$\texttt{PTRefiners}_{\texttt{R}}(P) \stackrel{\text{def}}{=} \{S \mid P \neq \texttt{PTSplit}_{\texttt{R}}(S,P) \ \wedge \ \exists \{B_i\}_i \subseteq P.\, S = \textstyle\bigcup_i B_i\}$$

$$\texttt{PT-Algorithm}_{\texttt{R}} : \begin{cases} \textbf{while } (P \text{ is not } \texttt{R}\text{-stable}) \textbf{ do} \\ \quad \textbf{choose } S \in \texttt{PTRefiners}_{\texttt{R}}(P); \\ \quad P := \texttt{PTSplit}_{\texttt{R}}(S,P); \\ \textbf{endwhile} \end{cases}$$

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

## 8. An algorithmic approach to declassification refinement

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

To understand how we can exploit these connections to get an algorithmic approach to the refinement of declassification policies, we note that stability corresponds to $\mathscr{B}$ completeness with respect to the post function of a given transition system (Ranzato and Tapparo 2005; Mastroeni 2008). On the other hand, both NI and DNI are characterised as $\mathscr{B}$ completeness problems with respect to 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, also allows us to refine declassification policies to make DNI hold.

The well-known Paige–Tarjan algorithm computes the coarsest bisimulation of a given partition as follows. Consider a relation $\texttt{R}$ such that $f = \widetilde{pre}(\texttt{R})$. The algorithm is given in Figure 4, where $P$ is a partition, $\texttt{PTSplit}_{\texttt{R}}(S,P)$ partitions each unstable block in $P$ with respect to $\texttt{R}$ with $B \cap f(S)$ and $B \smallsetminus f(S)$, while $\texttt{PTRefiners}_{\texttt{R}}(P)$ is the set of all the blocks in $P$, or obtained as the union of blocks in $P$ that make other blocks unstable. This algorithm has been shown to be a forward completeness problem 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 generalised to different input and output partitions (Mastroeni 2008), and, as a consequence, we can also trivially generalise the algorithm to different partitions in the input and output, respectively (see Section 7). The

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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 to ensure non-interference. Now we show that the same result can be obtained using the PT-algorithm on the partition induced by the declassification policy in the input, that is, $P_I = \{even, odd\}$, 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* with respect to the wlp semantics and to the output partition because

$$even \cap \ \mathit{Wlp}(H_a)_{|_{\mathtt{H}}} \neq \emptyset \ \wedge \ even \nsubseteq \mathit{Wlp}(H_a)_{|_{\mathtt{H}}}$$

where $a \neq 0$ and $\mathit{Wlp}(H_a)_{|_{\mathtt{H}}}$ represents the result of $\mathit{Wlp}(H_a)$ restricted to secret data. In fact, $H_a$, with $a \neq 0$ belongs to

$$\mathtt{PTRefiners}_{\mathtt{R}}(P_I) = \left\{ S \in P_O \;\middle|\; P_I \neq \mathtt{PTSplit}_{\mathtt{R}}(S, P_I) \ \wedge \ \exists \{B_i\}_i \subseteq P_O.\, S = \bigcup_i B_i \right\},$$

hence $P_I$ is refined by substituting the block $even$ with the new blocks

$$\{0\} = even \cap \ \mathit{Wlp}(H_a)_{|_{\mathtt{H}}}$$

and

$$even \smallsetminus \{0\} = even \smallsetminus \mathit{Wlp}(H_a)_{|_{\mathtt{H}}}.$$

At this point the input partition is stable and, indeed, is $P'_I = \{\{0\}, Even \smallsetminus \{0\}, odd\}$, which says that not only can we distinguish even secret inputs from the odd ones, but we can also distinguish between 0 and not 0 within the set of even numbers.

The next example again shows the refinement technique, but this time on the program fragment proposed in Zdancewic and Myers (2001), in order to show the relationship 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:

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

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 equality test between the password $p$ and the query $q$ fails). The public variables are $t, q, r$, so the partition induced by $\mathscr{H}$ is

$$\langle t, h, p, q, r\rangle \ \equiv \ \langle t', h', p', q', r'\rangle \text{ if and only if } t = t' \ \wedge q = q' \ \wedge \ r = r'.$$

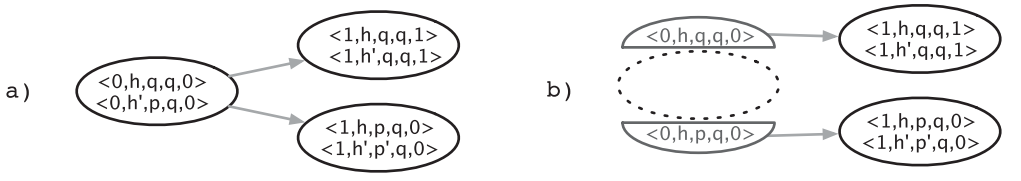
The above says we are considering two states that are L indistinguishable (as in ordinary NI). By checking completeness, we characterise the information that can be released. For example, consider the set of possible input states that are in the same equivalence class

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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 characterise *what* information is released, we complete the domain $\mathscr{H}$ by $\widetilde{pre}_P$ (below, right):

$$\langle 0, h, p, q, 0\rangle \mapsto \begin{cases} \langle 1, h, p, p, 1\rangle \\ \langle 1, h, p, q, 0\rangle \end{cases} \qquad \widetilde{pre}_P : \begin{cases} \langle 1, h, p, p, 1\rangle \mapsto \langle 0, h, p, p, 0\rangle \\ \langle 1, h, p, q, 0\rangle \mapsto \langle 0, h, p, q, 0\rangle \quad p \neq q \end{cases}$$

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 $p \neq q$, that is, we release the information whether $p = q$ or $p \neq q$.




In the picture above, we have represented the partition of states induced by the input and output observation (in this case they coincide). Note that this partition is not stable with respect to the function $\widetilde{pre}_P$, in particular, we have

$$\{\langle 0, h, p, q, 0\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\} \cap \widetilde{pre}_P(\{\langle 1, h, q, q, 1\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\}) \neq \emptyset \wedge$$
$$\{\langle 0, h, p, q, 0\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\} \not\subseteq \widetilde{pre}_P(\{\langle 1, h, q, q, 1\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\}).$$

Hence, the PT-algorithm splits the block $\{\langle 0, h, p, q, 0\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\}$ into the two new blocks

$$\{0, h, q, q, 0 \mid h, p \in \mathbb{V}^{\mathrm{H}}\} = \{\langle 0, h, p, q, 0\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\} \cap \widetilde{pre}_P(\{\langle 1, h, q, q, 1\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\})$$
$$\{0, h, p, q, 0 \mid h, p \in \mathbb{V}^{\mathrm{H}}, p \neq q\} = \{\langle 0, h, p, q, 0\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\} \setminus \widetilde{pre}_P(\{\langle 1, h, q, q, 1\rangle \mid h, p \in \mathbb{V}^{\mathrm{H}}\}).$$

This result corresponds exactly to what we would obtain by considering the completeness refinement. Moreover, this also corresponds to the result in Zdancewic and Myers (2001) obtained 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 non-interference 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

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

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, that is, 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, we have characterised, in the resulting abstract domain, the secure declassification policy.

## 9. Discussion

In this paper we have exploited the completeness of abstract interpretation for modelling non-interference for confidentiality policies based on declassification. Starting with Joshi and Leino's semantic formulation of NI (Joshi and Leino 2000), it is possible to characterise NI as a problem of $\mathscr{B}$-completeness for denotational semantics (Giacobazzi and Mastroeni 2005). This paper provides an equivalent formulation of NI as $\mathscr{F}$-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. $\mathscr{F}$-completeness formalises 'not exposing distinctions'. The advantage of our formalisation in comparison with 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 us to characterise an algorithmic approach for checking and refining declassification policies. This connection also leads 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 even allow us to consider infinite state systems. As future work, we are studying the practical use of these techniques when applied to more complex systems.

The relation between the abstract interpretation approach to NI (Giacobazzi and Mastroeni 2004; Giacobazzi and Mastroeni 2005) and many existing approaches for non-interference and declassification has been studied by means of examples (Hunt and Mastroeni 2005; Mastroeni 2005). Sabelfeld and Sands note that most existing proposals suffer from the lack of a compelling semantics for declassification. In earlier work they used the PER model (Sabelfeld and Sands 2001) to define selective dependency (Cohen 1977) by means of equivalence relations instead of abstract domains. They also show, through an example, that the PER model can be used to show that nothing more is learnt by an attacker than the policy itself releases (Sabelfeld and Sands 2007); in our model, we derive this formally (Corollary 5.2), and also show how counterexamples can be generated and the policy refined if a policy is not satisfied. Joshi and Leino (2000) introduced *abstract variables* to obtain a more general notion of security. In this case they substitute the secret variables with functions, that is, properties, of them. This corresponds

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

to abstract non-interference where we fix what we want to protect instead of what we allow to flow (Giacobazzi and Mastroeni 2004; Mastroeni 2005), so it is not helpful for computing what information is released. Darvas *et al.* (2005) used dynamic logic to analyse the declassification property dynamically. The information flow property is modelled as a dynamic logic formula. They then fix some declassifying preconditions and execute the analysis. If the analysis succeeds, there is an upper bound on the information disclosed; otherwise the precondition must be refined. Because of the connections between completeness and PT, our approach can provide a more systematic method for designing and refining these preconditions. Our approach differs from quantitative characterisations (Clark *et al.* 2004; Di Pierro *et al.* 2002) of the information released since we provide a *qualitative* analysis of the leaked secrets.

We were directly inspired by work on gradual release (Askarov and Sabelfeld 2007a). In addition to reasoning about the *where* dimension of declassification (as in gradual release), we can also reason about the *what* dimension of declassification. In recent work, and building directly on gradual release, Banerjee *et al.* (2008) showed how 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) does not show how to *compute* the information released by a program or to check whether the information released is more than that specified by the policy.

Unno *et al.* (2006) 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, we plan to investigate how their approach to NI can be combined with ours for DNI.

Alur *et al.* (2006) considered the preservation of secrecy under refinement and presented a simulation-based technique to show when one system is a refinement of another with respect to secrecy. They contend that their approach is flexible because it can express arbitrary secrecy requirements. In particular, if the specification does not maintain the secrecy of a property, 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, note that in this paper we have only discussed narrow non-interference, while in the original work on the weakening of non-interference by abstract interpretation another notion was introduced, namely, abstract non-interference (Giacobazzi and Mastroeni 2004). This notion was introduced for two reasons:

(a) Narrow non-interference is based on the computation of the *concrete semantics*, which can only be observed by an attacker.

https://doi.org/10.1017/S096012951100020X Published online by Cambridge University Press

(b) Narrow non-interference generates false alarms due to the abstract observation, so secure situations may be rejected as insecure.

In order to avoid false alarms and to allow an analysis of the program's semantics, abstract non-interference considers attackers that can compute the *abstract semantics* of programs, in other words, attackers that can statically analyse a program's code. The connection with completeness is also very strong for this notion (Giacobazzi and Mastroeni 2005), and we would like to investigate how we can generalise this connection to deal with different semantic and protection policies, exactly as we have done in this work for narrow non-interference.

The second direction we would like to pursue is to understand 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 to model active attackers as semantic transformers, and if we know the program points where the attacker can carry out its activity (Myers *et al.* 2004), we think that *Wlp* is a good semantic model for characterising both the 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 we can 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 interest for future work.

## Acknowledgements

This is a revised version of Banerjee *et al.* (2007), which appeared in the Proceedings of the Twenty-Third Conference on Mathematical Foundations of Programming Semantics (MFPS), 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 for improving the presentation.

## References

Alur, R., Cerny, P. and Zdancewic, S. (2006) Preserving secrecy under refinement. In: Proceedings of the International Colloquium on Automata, Languages and Programming (*ICALP '06*). *Springer-Verlag Lecture Notes in Computer Science* **4052** 107–118.

Amtoft, T., Bandhakavi, S. and Banerjee, A. (2006) A Logic for Information Flow in Object-Oriented Programs. In: *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages*, ACM Press 91–102.

Askarov, A. and Sabelfeld, A. (2007a) Gradual release: Unifying declassification, encryption and key release policies. In: *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society Press.

Askarov, A. and Sabelfeld, A. (2007b) 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*, ACM Press 53–60.

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

Banerjee, A., Naumann, D. A. and Rosenberg, S. (2008) Expressive declassification policies and modular static enforcement. In: *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society Press.

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

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

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

Cousot, P. (2002) Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science* **277** (1-2) 47–103.

Cousot, P. and Cousot, R. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT–SIGPLAN symposium on Principles of programming languages* (*POPL '77*), ACM Press 238–252.

Cousot, P. and Cousot, R. (1979) Systematic design of program analysis frameworks. In: *Proceedings of the 6th ACM SIGACT–SIGPLAN symposium on Principles of programming languages* (*POPL '79*), ACM Press 269–282.

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

Di Pierro, A., Hankin, C. and Wiklicky, H. (2002) Approximate non-interference. In: *Proceedings of the IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press 1–17.

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

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

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

Giacobazzi, R. and Quintarelli, E. (2001) Incompleteness, counterexamples and refinements in abstract model-checking. In: Cousot, P. (ed.) Proceedings of the 8th International Static Analysis Symposium (SAS'01). *Springer-Verlag Lecture Notes in Computer Science* **2126** 356–373.

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

Goguen, J. A. and Meseguer, J. (1984) Unwinding and inference control. In *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society Press 75–86.

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

Hunt, S. and Mastroeni, I. (2005) The PER model of abstract non-interference. In: Hankin, C. and Siveroni, I. (eds.) Proceedings of The 12th International Static Analysis Symposium (SAS '05). *Springer-Verlag Lecture Notes in Computer Science* **3672** 171–185.

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

Kahn, G. (1987) Natural semantics. In: Proceedings of Symposium on Theoretical Aspects of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **247** 22–39.

Li, P. and Zdancewic, S. (2005) Downgrading policies and relaxed noninterference. In: *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, ACM Press 158–170.

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

Mastroeni, I. (2008) Deriving bisimulations by simplifying partitions. In: Proceedings of the 9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08). *Springer-Verlag Lecture Notes in Computer Science* **4905** 147–171.

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

Myers, A. C., Chong, S., Nystrom, N., Zheng, L. and Zdancewic, S. (2001) Jif: Java information flow. Software release.

Myers, A. C., Sabelfeld, A. and Zdancewic, S. (2004) Enforcing robust declassification. In: *Proceedings IEEE Symposium on Security and Privacy*, IEEE Computer Society Press 21–34.

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

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

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

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

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

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

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

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

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

Zdancewic, S. and Myers, A. C. (2001) Robust declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press 15–23.