Proving Abstract Non-interference

Roberto Giacobazzi and Isabella Mastroeni

Dipartimento di Informatica
Università di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
(roberto.giacobazzi@ | mastroeni@sci.)univr.it

Abstract. In this paper we introduce a compositional proof-system for certifying abstract non-interference in programming languages. Certifying abstract non-interference means proving that no unauthorized flow of information is observable by the attacker from confidential to public data. The properties of the computation that an attacker may observe are specified as an abstract domain. Assertions specify the secrecy of a program relatively to the given attacker and the proof-system specifies how these assertions can be composed in a syntax-directed a la Hoare deduction of secrecy. We prove that the proof-system is sound relatively to the standard semantics of an imperative programming language. This provides a sound proof-system for both certifying secrecy in language-based security and deriving attackers which do not violate secrecy inductively on program's syntax.

Keywords: Abstract interpretation, language-based security, abstract non-interference, verification

1 Introduction

Standard non-interference has been introduced by Goguen and Meseguer in [19] as a key feature to model information flows in security. The idea behind non-interference in security is that users have given access control privileges and higher privileges are required in order to access files containing confidential data. In this way, when authorized users accessing public data are non-interfering with those on private resources, no leakage of confidential information is possible by observing public input/output behavior of the system. On this pattern most security polices are specified in language-based security, where users are program components specified in some high-level programming language (see [26]). Most methods and techniques for checking *secure information flows* in software, ranging from standard data-flow/control-flow analysis techniques to type inference, are based on non-interference. All of these approaches are devoted to prove that a system as a whole, or parts of it, does not allow confidential data to flow towards public variables. Type-based approaches are designed in such a way that well-typed programs do not leak secrets. In a security-typed language, a type is inductively associated at compile-time with program statements in such a way that any statement showing a potential flow disclosing secrets is rejected [28, 30, 32]. Similarly, data-flow/control-flow analysis techniques are devoted to statically discover flows of secret data into public variables [6, 22, 23, 27]. The problem of weakening non-interference,

J. Marcinkowski and A. Tarlecki (Eds.): CSL 2004, LNCS 3210, pp. 280–294, 2004.
© Springer-Verlag Berlin Heidelberg 2004

also known as refining security policies, has been recognized as a long standing major challenge in language-based security [26]. In standard non-interference, the attacker is able to fully analyze concrete computations. In this case, any conservative type/data-flow/control-flow analysis of information flows would discard all the programs which may provide any explicit or implicit concrete flows from confidential to public resources. Standard non-interference is therefore often too strict for any practical use in language-based security: most programs are rejected by static control/data flow analyzers or type checkers for non-interference. In order to adapt security policies to practical cases, it would be essential to know how much an attacker may learn from a program by (statically) analyzing its input/output behavior. This idea has recently lead to the definition of the notion of *abstract non-interference* [16]. Abstract non-interference captures a weaker form of non-interference, where non-interference is made parametric relatively to some abstract property of input/output behaviour. Consider the following program P written in a simple imperative language, where the **while**-statement iterates until x_1 is 0. Suppose $x_1 : \mathtt{H}$ is a secret variable and $x_2 : \mathtt{L}$ is a public variable:

$$\textbf{while}\ x_1\ \textbf{do}\ x_2 := x_2 * 2;\ x_1 := x_1 - 1\ \textbf{endw}$$

While in standard non-interference there is an implicit flow from x_1 to x_2, due to the **while**-statement, because x_2 changes depending on the initial value of x_1, this is not true for weaker abstractions of public data. In particular if the attacker can only observe the property of being power of 2 of public variables (x_2), since the operation cannot change its status of being or not a power of two. Then an attacker is unable to observe any interference due to the implicit flow. Abstract non-interference generalizes this idea to arbitrary abstractions of the semantics of a programming language. This provides both a characterization of the degree of secrecy of a program relatively to what an attacker can analyze about its input/output information flow and the possibility for certifying code relatively to some weaker form of non-interference.

The Problem

Abstract non-interference is based on the idea that the model of an attacker is an abstract interpretation of the semantics of the program. A program satisfies the abstract non-interference condition relatively to some given abstraction (attacker) if the abstraction obfuscates any possible interference between confidential and public data. In [16] the authors introduce a step-by-step weakening of Goguen and Meseguer's non-interference by specifying abstract non-interference as a property of the semantics of the program. The idea of modeling attackers as abstract domains provides advanced methods for deriving attackers by systematically transforming the corresponding abstract domains. An algebraic characterization of the most precise secure attacker, i.e., the most precise abstraction for which the program satisfies the abstract non-interference property, is given as a fixpoint domain construction. This abstraction, as well as any abstractions for which the program satisfies abstract non-interference, is both a model of an harmless attacker and a certificate for the security degree of the program. However the original definition of abstract non-interference is not specified inductively on program's syntax but rather it is derived as an abstraction of the concrete semantics of the

whole program. This makes the use of abstract non-interference hard in automatic program certification mechanisms, such as in proof-carrying code architectures [24] and in type-based verification algorithms. The logical approach to secure information flow is not new. In [12] dynamic logic is used for characterizing secure information flows, deriving a theorem prover for checking programs. In [1] an axiomatic approach for checking secure information flows is provided. In particular the authors syntactically derive the secure information flows that may happen during the execution. Both these works don't characterize the power of the attacker.

Main Contribution

In this paper we introduce a compositional proof-system for certifying abstract non-interference in programming languages which means proving that the program satisfies an abstract non-interference constraint relatively to some given abstraction of its input/output. Abstractions are specified in the standard abstract interpretation [9] framework. Assertions in the proof-system have the form of Hoare triples: $(\eta)P(\rho)$ where P is a program fragment and η and ρ are abstractions of program's data. However the interpretation of abstract non-interference assertions is rather different from partial correctness assertions (see [3]): $(\eta)P(\rho)$ means that P is unable to disclose secrets if input and output values on public variables are approximated respectively in η and ρ. Hence, abstract non-interference assertions specify the secrecy of a program relatively to a given model of an attacker and the proof-system specifies how these assertions can be composed in a syntax-directed *a la* Hoare deduction of secrecy. We introduce two proof-systems for checking abstract non-interference. The first deals with a stronger notion of abstract non-interference called *narrow abstract non-interference* [16]. The advantage of narrow abstract non-interference is in the simplicity of the proof-system and in its natural derivation from the operational semantics of the language. This proof-system is necessary in order to derive a proof-system for most general abstract non-interference assertions. We prove that the proof-systems are sound relatively to the standard semantics of an imperative programming . Both proof-systems provide a deeper insight in abstract non-interference, by specifying how assertions concerning secrecy compose with each other. This is essential for any static semantics for secrecy devoted to derive certificates specifying the degree of secrecy of a program.

2 Basic Notions

If S and T are sets, then $\wp(S)$ denotes the powerset of S, $S\smallsetminus T$ denotes the set-difference between S and T, $S \subsetneq T$ denotes strict inclusion, and for a function $f : S \to T$ and $X \subseteq S$, $f(X) \stackrel{\text{def}}{=} \{f(x) \mid x \in X\}$. We will often denote $f(\{x\})$ as $f(x)$. $\langle P, \leq\rangle$ denotes a poset P with ordering relation $\leq$, while $\langle P, \leq, \vee, \wedge, \top, \bot\rangle$ denotes a complete lattice P, with ordering $\leq$, *lub* $\vee$, *glb* $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$. Often, $\leq_P$ will be used to denote the underlying ordering of a poset P, and $\vee_P$, $\wedge_P$, $\top_P$ and $\bot_P$ denote the basic operations and elements if P is a complete lattice. $f : C \to A$ is (completely) additive if f preserves *lub*'s of all subsets of C (emptyset included). A proof-system $\mathcal{P}$ on a set of formulas $\varPhi$ is a finite set of axiom schemes

and proof rules. A proof of φ in $\mathcal{P}$ is a finite sequence of formulas $\varphi_1, \dots, \varphi_n$ such that $\varphi = \varphi_n$ and each φ_i is either an axiom in $\mathcal{P}$ or it can be obtained by applying a proof rule in $\mathcal{P}$. In this case φ is also called a *theorem* of $\mathcal{P}$, and denoted $\vdash_{\mathcal{P}} \varphi$.

2.1 Abstract Interpretation

Abstract domains can be equivalently formulated either in terms of Galois connections or closure operators [10]. An *upper closure operator* on a poset P is an operator $\rho : P \to P$ monotone, idempotent and extensive ($\forall x \in P.\ x \leq_P \rho(x)$). The set of all upper closure operators on P is denoted by *uco*(P). Let $\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ be a complete lattice. Closure operators are uniquely determined by the set of their fix-points $\rho(C)$. $\rho(C)$ is a complete sub-lattice of C iff ρ is additive. If C is a complete lattice then *uco*(C) ordered point-wise is also a complete lattice, denoted by $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, \lambda x.x \rangle$, where for every $\rho, \eta \in uco(C)$, $\{\rho_i\}_{i \in I} \subseteq uco(C)$ and $x \in C$: $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$; $(\sqcap_{i \in I} \rho_i)(x) = \wedge_{i \in I} \rho_i(x)$; and $(\sqcup_{i \in I} \rho_i)(x) = x \;\Leftrightarrow\; \forall i \in I.\ \rho_i(x) = x$. The *disjunctive completion* of a domain is the most abstract domain able to represent the concrete disjunction of its objects: $\curlyvee(\rho) = \sqcup \{\eta \in uco(C) | \eta \sqsubseteq \rho$ and η is additive$\}$. ρ is disjunctive iff $\curlyvee(\rho) = \rho$ (cf. [10, 18]). Closure operators and partitions are related concepts. A closure $\eta \in uco(\wp(X))$ induces a partition on X: $\{\, [x]_\eta \mid x \in X \,\}$, where $[x]_\eta \overset{\text{def}}{=} \{\, y \mid \eta(x) = \eta(y) \,\}$. The most concrete closure that induces the same partition of values as η is $\Pi(\eta) \overset{\text{def}}{=} \curlyvee(\{\, [x]_\eta \mid x \in X \,\})$ (see Fig. 1). The idea is that $\Pi(\eta)$ is the

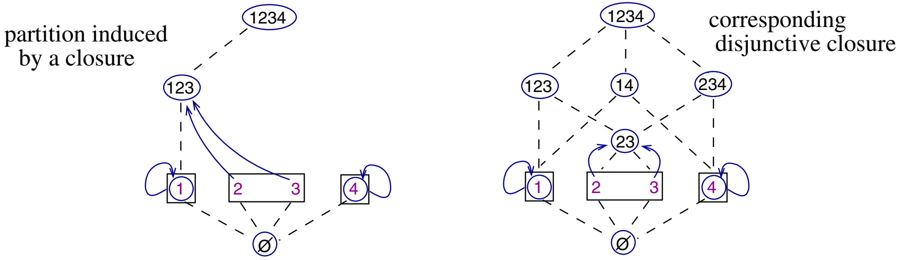

Fig. 1. Example of partitions as disjunctive completion

most concrete closure such that for any $y \in \Pi(\eta(x))$: $\Pi(\eta(x)) = \Pi(\eta(y))$, while in general $\eta(y) \subseteq \eta(x)$.

2.2 The Deterministic Language

In the following we consider a simple imperative language, IMP [31] where programs are commands with the following syntax:

$$c ::= \mathbf{nil} \mid x := e \mid c; c \mid \mathbf{while}\ x\ \mathbf{do}\ c\ \mathbf{endw}$$

with e denoting expressions evaluated in the set of values $\mathbb{V}$ with standard operations, i.e., if $\mathbb{V} = \mathbb{Z}$ then e can be any arithmetical expression. As usual, $\mathbb{V}$ can be structured as a flatdomains, where bottom element, $\bot$, denotes the value of undefined vari-

ables. In the following we will denote by $\mathit{Var}(P)$ the set of variables of the program $P \in$ IMP. Let's consider the well-known operational semantics of IMP [31]. The operational semantics naturally induces a transition relation on a set of states Σ, denoted $\rightarrow$, specifying the relation between a state and its possible successors. $\langle \Sigma, \rightarrow \rangle$ is a transition system. In our case, if $|\mathit{Var}(P)| = n$ then $\Sigma = \mathbb{V}^n$. We follow Cousot's construction [8, 11], defining semantics, at different levels of abstractions, as the abstract interpretation of the maximal trace semantics of a transition system associated with each well-formed program. In the following, Σ^+ and $\Sigma^\omega \stackrel{\text{def}}{=} \mathbb{N} \longrightarrow \Sigma$ denote respectively the set of finite nonempty and infinite sequences of symbols in Σ. Given a sequence $\sigma \in \Sigma^\infty \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$, its length is denoted $|\sigma| \in \mathbb{N} \cup \{\omega\}$ and its i-th element is denoted σ_i. A non-empty finite (infinite) *trace* $\sigma \in \Sigma^\infty$ is a finite (infinite) sequence of program states where two consecutive elements are in the transition relation $\rightarrow$, i.e., for all $i < |\sigma|$: $\sigma_i \rightarrow \sigma_{i+1}$. The *maximal trace semantics* [11] of a transition system associated with a program P is $[\![P]\!]^\infty \stackrel{\text{def}}{=} [\![P]\!]^+ \cup [\![P]\!]^\omega$, where if $T \subseteq \Sigma$ is a set of final/blocking states then $[\![P]\!]^{\dot{n}} = \{\sigma \in \Sigma^+ | |\sigma| = n, \forall i \in [1, n) \,.\, \sigma_{i-1} \rightarrow \sigma_i\}$, $[\![P]\!]^\omega = \{\sigma \in \Sigma^\omega | \, \forall i \in \mathbb{N} \,.\, \sigma_i \rightarrow \sigma_{i+1}\}$, $[\![P]\!]^+ = \cup_{n>0}\{\sigma \in [\![P]\!]^{\dot{n}} | \, \sigma_{n-1} \in T\}$, and $[\![P]\!]^n = [\![P]\!]^{\dot{n}} \cap [\![P]\!]^+$. If $\sigma \in [\![P]\!]^+$, then $\sigma_\dashv$ and $\sigma_\vdash$ denote respectively the final and initial state of σ. The *denotational semantics* associates input/output functions with programs, by modeling non-termination by $\perp$. This semantics is derived in [8] by abstract interpretation from the maximal trace semantics with abstraction $\alpha^{\mathcal{D}}(X) \stackrel{\text{def}}{=} \lambda s \in \Sigma.\, \{\sigma_\dashv | \sigma \in X \cap \Sigma^+,\ s = \sigma_\vdash\} \cup \{\, \perp \,|\, \exists \sigma \in X \cap \Sigma^\omega,\ s = \sigma_\vdash \,\}$. Note that, since our programs are deterministic, $\alpha^{\mathcal{D}}(X)(s)$ is always a singleton. It is well known that we can associate with each program $P \in$ IMP a function $[\![P]\!]$ denoting its input/output relation, such that $[\![P]\!] \stackrel{\text{def}}{=} \alpha^{\mathcal{D}}([\![P]\!]^\infty)$.

3 Non-interference

Many security problems in language-based security are problems of interference. In order to keep some data confidential, a user might state a policy stipulating that no data visible to other users is affected by modifying confidential data. This policy allows programs to manipulate and modify private data, as long as visible outputs of those programs do not reveal information about the private data. A policy of this sort is a *non-interference* policy [19], also referred as *secrecy* [29]. Confidential data are considered *private*, labeled with H (high-level of secrecy), while all other data are public, labeled with L (low-level of secrecy) [14]. Secrecy is usually formulated saying that the *final* value of a low variable does not depend on the *initial* value of high-variables [29]. An attacker (or unauthorized user) is assumed to be allowed to view only information that is not secret. The usual method for showing that secrecy holds is to verify that the attacker cannot observe *any* difference between two executions that differ only in their secret input [29, 20]. In this case we say that the program has only *secure information flows* [22, 29, 14, 4, 5, 7, 13]. In order to model this situation we consider the denotational semantics $[\![P]\!]$ of the program P. We consider a typing function $t \in \mathit{Var} \longrightarrow \{\mathtt{H}, \mathtt{L}\}$, which associates with each variable in a program its security class. In the following, if $x \in \mathit{Var}(P)$ then we denote $x : t(x)$ the corresponding security typing. If $\mathtt{T} \in \{\mathtt{H}, \mathtt{L}\}$, $v \in \mathbb{V}^n$, and $n = |\{x \in \mathit{Var}(P) | t(x) = \mathtt{T}\}|$, we abuse notation by denoting $v \in \mathbb{V}^{\mathtt{T}}$ the

fact that v is a possible value for the vector of variables with security type $\mathtt{T}$. Moreover, we assume that any state $s \in \Sigma$ can be seen as a pair $\langle h, l\rangle$ where $h \in \mathbb{V}^{\mathtt{H}}$ and $l \in \mathbb{V}^{\mathtt{L}}$ and we denote the projection on low values as $\langle h, l\rangle^{\mathtt{L}} = l$. In this case, *standard non-interference* can be formulated as follows.

A program P is *secure* if
$$\forall v \in \mathbb{V}^{\mathtt{L}}, \forall v_1, v_2 \in \mathbb{V}^{\mathtt{H}} \,.\, (\llbracket P\rrbracket(v_1, v))^{\mathtt{L}} = (\llbracket P\rrbracket(v_2, v))^{\mathtt{L}}$$

In [16] we introduced a weaker notion of non-interference modeling weaker information flows. The idea is that an attacker can observe only some properties of public concrete values. The observable properties are modeled as abstractions. As usual in abstract interpretation, a property is an *upper closure operator* on the concrete domain of computation [10]. It is clear that, any observation made on program input/output behaviour by abstract interpretation of its semantics strictly depends upon the chosen abstract domains. The *model of an attacker*, also called *attacker*, is therefore a pair of abstractions: $\langle \eta, \rho\rangle$, with $\eta, \rho \in \mathit{uco}(\wp(\mathbb{V}^{\mathtt{L}}))$, representing what an observer can see about respectively the input and output of a program. Given a program P, *narrow (abstract) non-interference*, denoted $[\eta]P(\rho)$, and *abstract non-interference*, denoted $(\eta)P(\rho)$, introduced in [16], represent a weakening of standard non-interference relatively to a given model of an attacker $\langle \eta, \rho\rangle$. In the following we will abuse notation by denoting with $\llbracket P\rrbracket$ also the additive lifting of $\llbracket P\rrbracket$ to sets of states. Moreover we will use the following simplified notation, $(\llbracket P\rrbracket(h_1, l_1))^{\mathtt{L}} = \llbracket P\rrbracket(h_1, l_1)^{\mathtt{L}}$.

Definition 1. *Let* $\eta, \rho \in \mathit{uco}(\wp(\mathbb{V}^{\mathtt{L}}))$. *A program* $P \in \textsc{Imp}$ *is such that* $[\eta]P(\rho)$ *if* $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}} \,.\, \eta(l_1) = \eta(l_2) \;\Rightarrow\; \rho(\llbracket P\rrbracket(h_1, l_1)^{\mathtt{L}}) = \rho(\llbracket P\rrbracket(h_2, l_2)^{\mathtt{L}})$. $P \in \textsc{Imp}$ *is such that* $(\eta)P(\rho)$ *if* $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l \in \mathbb{V}^{\mathtt{L}} \,.\, \rho(\llbracket P\rrbracket(h_1, \eta(l))^{\mathtt{L}}) = \rho(\llbracket P\rrbracket(h_2, \eta(l))^{\mathtt{L}})$.

The difference between abstract and narrow non-interference lies upon what the attacker may observe of the input property η. Due to the possible presence of *deceptive flows* in narrow non-interference (see [16]), abstract non-interference represents a weaker notion.

Proposition 1. $[\mathtt{id}]P(\mathtt{id}) \;\Rightarrow\; (\eta)P(\rho) \qquad [\eta]P(\rho) \;\Rightarrow\; (\eta)P(\rho)$

Example 1. Let $\mathit{Sign} = \{\mathbb{Z}, +, -, \varnothing\}$ and $\mathit{Par} = \{\mathbb{Z}, 2\mathbb{Z}, 2\mathbb{Z}+1, \varnothing\}$, and consider the program $P \stackrel{\text{def}}{=} l := 2{*}l{*}h^2$ with security typing $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V} = \mathbb{Z}$. Note that $\mathit{Par}(-2) = \mathit{Par}(4) = 2\mathbb{Z}$, but $\mathit{Sign}(\llbracket P\rrbracket(h, -2)^{\mathtt{L}}) = 0- \neq 0+ = \mathit{Sign}(\llbracket P\rrbracket(h, 4)^{\mathtt{L}})$. Namely $\not\models [\mathit{Par}]P(\mathit{Sign})$ due to a deceptive flow generated by a change of low inputs having the same property in Sign.

In [16] two methods for deriving the most concrete output observation for a program, given the input one, for both narrow and abstract non-interference are provided. In particular the idea is that of collecting in the same abstract object all the elements that, if distinguished, would generate a visible flow. This most concrete output observation that is not able to get information from the program P observing η in input for narrow and abstract non-interference is respectively denoted $[\eta]\llbracket P\rrbracket(\mathtt{id})$ and $(\eta)\llbracket P\rrbracket(\mathtt{id})$.

Theorem 1 ([16]). $[\eta][\![P]\!](\mathtt{id}) \sqsubseteq \rho \Leftrightarrow [\eta]P(\rho)$ *and* $(\eta)[\![P]\!](\mathtt{id}) \sqsubseteq \rho \Leftrightarrow (\eta)P(\rho)$.

In the following whenever ρ is such that $(\eta)[\![P]\!](\mathtt{id}) \sqsubseteq \rho$ we will write $\models (\eta)P(\rho)$. The same holds for the narrow non-interference.

The main limitation on the use of either $[\eta][\![P]\!](\mathtt{id})$ or $(\eta)[\![P]\!](\mathtt{id})$ for checking abstract non-interference is due to their dependence from the final result of the concrete semantics of the program itself. This makes the construction of $(\eta)[\![P]\!](\mathtt{id})$ and $[\eta][\![P]\!](\mathtt{id})$ a hard task for large programs. In particular, no evidence is made in [16] on how these abstract domains can be derived inductively on program's syntax. This problem is solved in the next section, where a proof-system is introduced for both narrow and abstract non-interference.

4 Axiomatic Abstract Non-interference

In this section we introduce a proof-system for certifying abstract non-interference of programs. We assume a set Φ of basic formulas which can be freely generated from some given set of predicates on $\mathbb{V}^{\mathrm{L}}$ with the basic connectives $\wedge$, $\vee$ and $\neg$. An abstract domain $\rho \in uco(\wp(\mathbb{V}^{\mathrm{L}}))$ can therefore be represented as a $\wedge$-closed set of formulas in Φ. $\rho = \bigvee(\rho)$, i.e., ρ is disjunctive, iff it is closed under $\vee$ [18]. Note also that $\rho = \Pi(\rho)$ iff ρ is closed both by $\vee$ and $\neg$ (cf. [17, 25]). The semantics of a set of formulas is the corresponding abstract domain. The interpretation of $\bigsqcap$ and $\bigsqcup$ are therefore straightforward. As in most programming languages, IMP allows both explicit (through assignment) and implicit (through conditionals) flows [13]. The source of implicit flows in IMP is the **while**-statement.

In order to certify secrecy when implicit flows may occur, we need to model the properties that are invariant during the executions of programs. Intuitively an abstraction is invariant for a program fragment P, written $\{\rho\}_{\mathrm{L}}\ P\ \{\rho\}_{\mathrm{L}}$, when by observing the property ρ of public input of P, we are not able to observe any differences in the ρ property of the public output. In other words $\{\rho\}_{\mathrm{L}}\ P\ \{\rho\}_{\mathrm{L}}$ means that P is observably equivalent to **nil** as regards as the property ρ. This information is essential in order to certify the lack of implicit flows relatively to an abstraction. These invariant abstractions are obtained with an *a la* Hoare proof-system, where assertions are invariant properties of the form $\{\rho\}_{\mathrm{L}}\ P\ \{\rho\}_{\mathrm{L}}$, with $\rho \in uco(\wp(\mathbb{V}^{\mathrm{L}}))$. Invariants of expressions are parametric on a public variable: $\models \{\rho\}\ \langle e, x\rangle\ \{\rho\}$ iff $\forall l \in \mathbb{V}^{\mathrm{L}}, \forall h \in \mathbb{V}^{\mathrm{H}} . \rho(\mathcal{E}[\![e]\!](h, l)) = \rho(l_{|_x})$, where for any expression e, $\mathcal{E}[\![e]\!] : \Sigma \longrightarrow \mathbb{V}$ is the standard semantics of expressions and $l_{|_x}$ denotes the value that in $l \in \mathbb{V}^{\mathrm{L}}$ is assigned to x. The intuition is that e does not change the property ρ of the value of x. The public invariants of programs are defined as $\models \{\rho\}_{\mathrm{L}}\ P\ \{\rho\}_{\mathrm{L}}$ iff $\forall l \in \mathbb{V}^{\mathrm{L}}$, $\forall h \in \mathbb{V}^{\mathrm{H}} . \rho([\![P]\!](h, l)^{\mathrm{L}}) = \rho(l)$. Public invariants for programs can be derived by induction on the syntax of IMP by using the proof-system $\mathcal{I} = \{\mathbf{I1}, \ldots, \mathbf{I8}\}$ as defined in Table 1. Rule **I1** says that the property $\top$, which is unable to distinguish any value, is invariant for any program. **I2** says that any property is invariant for the program **nil**. The same holds if the program is an assignment to high variables (**I3**), since by definition invariants are constraints on low variables only. In **I4** if a property is invariant for the evaluation of an expression as regards as the input value of a low variable x, then it is invariant for the assignment to x. Consider for example

Table 1. Derivation of public invariants of programs

$$\textbf{I1: } \{\top\}_{\mathtt{L}}\, c\, \{\top\}_{\mathtt{L}} \quad \textbf{I2: } \{\rho\}_{\mathtt{L}}\, \textbf{nil}\, \{\rho\}_{\mathtt{L}} \quad \textbf{I3: } \frac{x : \mathtt{H}}{\{\rho\}_{\mathtt{L}}\, x := e\, \{\rho\}_{\mathtt{L}}} \quad \textbf{I4: } \frac{\{\rho\}\, \langle e, x\rangle\, \{\rho\},\ x : \mathtt{L}}{\{\rho\}_{\mathtt{L}}\, x := e\, \{\rho\}_{\mathtt{L}}}$$

$$\textbf{I5: } \frac{\{\rho\}_{\mathtt{L}}\, c_1\, \{\rho\}_{\mathtt{L}},\ \{\rho\}_{\mathtt{L}}\, c_2\, \{\rho\}_{\mathtt{L}}}{\{\rho\}_{\mathtt{L}}\, c_1; c_2\, \{\rho\}_{\mathtt{L}}} \quad \textbf{I6: } \frac{\{\rho\}_{\mathtt{L}}\, c\, \{\rho\}_{\mathtt{L}}}{\{\rho\}_{\mathtt{L}}\, \textbf{while } x \textbf{ do } c \textbf{ endw}\, \{\rho\}_{\mathtt{L}}} \quad \textbf{I7: } \frac{\{\rho'\}_{\mathtt{L}}\, c\, \{\rho'\}_{\mathtt{L}},\ \rho' \sqsubseteq \rho}{\{\rho\}_{\mathtt{L}}\, c\, \{\rho\}_{\mathtt{L}}}$$

Table 2. Axiomatic narrow (abstract) non-interference

$$\textbf{N0: } \frac{[\eta][\![c]\!](\mathtt{id}) \sqsubseteq \rho}{[\eta]c(\rho)} \quad \textbf{N1: } [\eta]c(\top) \quad \textbf{N2: } \frac{\Pi(\eta) \sqsubseteq \Pi(\rho)}{[\eta]\textbf{nil}(\rho)} \quad \textbf{N3: } \frac{[\eta]e(\rho),\ [\Pi(\eta) \sqsubseteq \Pi(\rho)],\ x : \mathtt{L}}{[\eta]x := e(\rho)}$$

$$\textbf{N4: } \frac{x : \mathtt{H},\ \Pi(\eta) \sqsubseteq \Pi(\rho)}{[\eta]x := e(\rho)} \quad \textbf{N5: } \frac{[\eta]c_1(\rho),\ [\rho]c_2(\beta)}{[\eta]c_1; c_2(\beta)} \quad \textbf{N6: } \frac{\{\rho\}_{\mathtt{L}}\, c\, \{\rho\}_{\mathtt{L}}}{[\rho]\textbf{while } x \textbf{ do } c \textbf{ endw}(\rho)}$$

$$\textbf{N7: } \frac{[\eta']c(\rho'),\ \eta \sqsubseteq \eta',\ \rho' \sqsubseteq \rho}{[\eta]c(\rho)} \quad \textbf{N8: } \frac{\forall i \in I\, .\, [\eta]c(\rho_i)}{[\eta]c(\bigsqcup_{i\in I} \rho_i)} \quad \textbf{N9: } \frac{\forall i \in I\, .\, [\eta]c(\rho_i)}{[\eta]c(\bigsqcap_{i\in I} \rho_i)}$$

the expression $l + 2$, then the property $Sign$ (which abstracts on the sign of an integer variable) is not invariant, since if we consider the input value $l = -1$, then we have that $Sign(l + 2) = Sign(1) = + \neq Sign(l) = -$. On the other hand, we have that Par (which abstract on the parity of an integer variable) is invariant for this expression as regards as the variable l, since the operation $l + 2$ doesn't change the parity of the value assigned to l. At this point if the statement is $l := l + 2$, then we have that $\{Par\}_{\mathtt{L}}\, l := l + 2\, \{Par\}_{\mathtt{L}}$. Note that, in order to apply this rule, if $\mathbb{V}^{\mathtt{L}} = \mathbb{V}_1 \times \ldots \times \mathbb{V}_n$, then $\rho \in uco(\mathbb{V}^{\mathtt{L}})$ is such that $\rho(\langle x_1, \ldots, x_n\rangle) = \langle \rho(x_1), \ldots, \rho(x_n)\rangle$. Rule **I5** says that the invariants distribute on the sequential composition. **I6** states that, given a **while**-statement, if a property is invariant for the body, then the same property is invariant for the whole statement. This rule holds since the only modifications of variables made by the **while**, are made by its body. Weakening (**I7**) says that any more abstract property of an invariant is still invariant. A derivation in the proof-system of public invariants in Table 1 is denoted $\vdash_{\mathcal{I}}$.

Theorem 2. *Let* $P \in \textsc{Imp}$ *and* $\rho \in uco(\mathbb{V}^{\mathtt{L}})$. *If* $\vdash_{\mathcal{I}} \{\rho\}_{\mathtt{L}}\, P\, \{\rho\}_{\mathtt{L}}$ *then* $\models \{\rho\}_{\mathtt{L}}\, P\, \{\rho\}_{\mathtt{L}}$.

We can now introduce a proof-system for narrow abstract non-interference. This is specified as in Table 2. Rule **N0** derives from Th. 1. It states that given a program c and an input observation η we can derive the most concrete output observation that makes the program secret. This corresponds to finding the strongest post-condition (viz. the most concrete abstract domain) for the program c with precondition η such that narrow abstract non-interference holds. This is a "semantic rule", because it involves the construction of the abstract domain $[\eta][\![c]\!](\mathtt{id})$, which is equivalent to compute the concrete

semantics of the command c [16]. However, this rule allows us to include in the narrow abstract non-interference proofs, also assertions which can be systematically derived as an abstract domain transformation as shown in [16]. Rules **N1** says that if the output observation is the property $\top$, then the input can be any property. **N2** says that **nil** is secret for any possible attacker such that the partition induced by input observation is more concrete than the output one. This condition is necessary since in this case abstract non-interference corresponds to saying $\forall l_1, l_2 \,.\, \eta(l_1) = \eta(l_2) \;\Rightarrow\; \rho(l_1) = \rho(l_2)$ which holds iff $\Pi(\eta) \sqsubseteq \Pi(\rho)$. Rule **N3** considers a notion of secrecy extended to expressions, i.e., $\models [\eta]e(\rho)$ iff $\forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}} \,.\, \eta(l_1) = \eta(l_2)$ we have $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}} \,.\, \rho(\mathcal{E}[\![e]\!](h_1, l_1)) = \rho(\mathcal{E}[\![e]\!](h_2, l_2))$. Being the variable public, the secrecy of the expression distributes on the assignment when the partition induced by the input observation is more concrete than the output one. This condition on the induced partitions is necessary only when there are public variables for which the assignment behaves as **nil** (see **N2**). Rule **N4** says that an assignment to a high variable is always secret when the partition induced by input observation is more concrete than the output one since an assignment to private variables behaves as **nil** for the public variables. Indeed note that if, for example, we have the statement $h := h + 1$, then clearly $\rho([\![h := h + 1]\!](h, l_1)^{\mathtt{L}}) = \rho(l_1)$ and $\rho([\![h := h + 1]\!](h, l_2)^{\mathtt{L}}) = \rho(l_2)$. This means that also in this case narrow non-interference corresponds to saying $\eta(l_1) = \eta(l_2) \;\Rightarrow\; \rho(l_1) = \rho(l_2)$. Both **N3** and **N4** consider closures on tuples that are tuples of abstractions, as in **I4**. Rule **N5** shows how we can compose the attackers in presence of sequential composition of programs. In particular two programs c_1 and c_2 compose secretly when c_1 is secret for the output observation which is the input one that makes c_2 secret. **N6** controls the **while**-statement. In particular the condition $\{\rho\}_{\mathtt{L}}\ c\ \{\rho\}_{\mathtt{L}}$ states that the program c is not acting on the property ρ of the public data, namely ρ is invariant in the execution of c, in the sense that the property ρ of public data is not changed by the execution of c. If this happens then the behaviour of c observed from ρ is the same as the program **nil**, and therefore the fact that the **while** is executed or not is not distinguishable from an observer. We apply this rule also when the guard is a low variable, because narrow non-interference may observe also deceptive flows. **N7** is the consequence rule, which states that we can concretize the input observation and we can abstract the output one, as observed in [16]. Finally **N8** and **N9** says that both the least upper bound and the greatest lower bound of output observations making a program secret, still make the program secret. We denote by $\mathcal{N} = \mathcal{I} \cup \{\mathbf{N0}, \ldots, \mathbf{N9}\}$ the proof-system for narrow abstract non-interference and by $\mathcal{N}_0 = \mathcal{I} \cup \{\mathbf{N1}, \ldots, \mathbf{N9}\}$ the same proof system without the semantic rule **N0**. Next result specifies that the proof-system, without the rule **N0**, is sound.

Theorem 3. *Let $P \in$ IMP and $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$. If $\vdash_{\mathcal{N}_0} [\eta]P(\rho)$ then $\models [\eta]P(\rho)$.*

Example 2. Consider the closure Par which observes parity, and the program:

$$P \stackrel{\text{def}}{=} l := 2 * h;\ \textbf{while}\ h\ \textbf{do}\ l := l + 2;\ h := h - 1\ \textbf{endw}$$

with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L} \rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$. We have $[\top]2 * h(\rho_1)$ where ρ_1 is the closure which is not able to distinguish even numbers, i.e., $\rho_1 = \curlyvee(\{2\mathbb{Z}\} \cup \{\ \{n\} \,|\, n \text{ odd}\ \})$. Therefore, by **N3**, we obtain $[\top]l := 2 * h(\rho_1)$ (note that since there is only one low variable we ignore the condition $\Pi(\eta) \sqsubseteq \Pi(\rho)$). Consider

now the **while**-statement. We note that the operation $l + 2$ leaves unchanged the parity of l, this means that if the input is even the the output is even, and similarly if it is odd. Namely for each n such that $Par(n) = Par(l)$ then $Par(\mathcal{E}[\![l + 2]\!](h, n)) = Par(n + 2) = Par(n) = Par(l)$. Therefore $\{Par\}\ \langle l + 2, l\rangle\ \{Par\}$ which implies

$$\frac{\{Par\}\ \langle l+2, l\rangle\ \{Par\}}{\{Par\}_{\mathtt{L}}\ l := l + 2\ \{Par\}_{\mathtt{L}}} \qquad \frac{h : \mathtt{H}}{\{Par\}_{\mathtt{L}}\ h := h + 1\ \{Par\}_{\mathtt{L}}}$$

Therefore, by **I5**, we have that $\{Par\}_{\mathtt{L}}\ l := l + 2;\ h := h - 1\ \{Par\}_{\mathtt{L}}$. Now we can apply rule **N6** obtaining

$$\frac{\{Par\}_{\mathtt{L}}\ l := l + 2;\ h := h - 1\ \{Par\}_{\mathtt{L}}}{[Par]\textbf{while}\ h\ \textbf{do}\ l := l + 2;\ h := h - 1\ \textbf{endw}(Par)}$$

Finally note that $\rho_1 \sqsubseteq Par$ hence by **N7** we have also that $[\top]l := 2 * h(Par)$, therefore we can apply rule **N5** and we obtain that $[\top]P(Par)$.

Unfortunately the system $\mathcal{N}_0$ is not complete, and in particular **N5** is the rule that introduces incompleteness.

Example 3. Consider the property Par observing parity, and the following program P with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$.

$$P \stackrel{\text{def}}{=} l := 4 * h^2 + 4;\ \textbf{while}\ h\ \textbf{do}\ l := l\ \mathtt{mod}\ 4;\ h := 0\ \textbf{endw}$$

Let us denote $c \stackrel{\text{def}}{=} \textbf{while}\ h\ \textbf{do}\ l := l\ \mathtt{mod}\ 4;\ h := 0\ \textbf{endw}$. We can prove that we have $\models [\top]l := 4h^2 + 4(\rho_1)$, where ρ_1 is defined in Example 2, and $\models [\top]P(\rho_1)$. But we can show that we have $\not\models [\rho_1]c(\rho_1)$ since $\rho_1([\![c]\!](0,5)^{\mathtt{L}}) = 5 \neq \rho_1([\![c]\!](1,5)^{\mathtt{L}}) = 1$. This means that $\not\vdash_{\mathcal{N}\smallsetminus\{\mathbf{N1}\}} [\top]P(\rho_1)$.

It is clear that rule **N0** makes the proof system complete. This is a straight consequence of Theorem 1.

Corollary 1. *The proof-system $\mathcal{N}$ is complete.*

We now introduce in Table 3 a proof-system for abstract non-interference, i.e., modeling how $(\eta)P(\rho)$ assertions compose inductively on program's syntax. The rules **A0** and **A1** in Table 3 are similar to the ones in Table 2. The rule **A2** differs from **N2** since abstract non-interference avoids deceptive flows. In rule **A3** we consider the generalization of the notion of abstract non-interference to expressions as we made for the narrow one. Moreover, as in **N3**, we consider here only abstractions of tuples that are tuples of abstractions. **A4** is straightforward.In order to understand the difference from **N4** consider the example used for explaining **N4**, i.e., $h := h + 1$ then in abstract non-interference we compute the following sets: $\rho([\![h := h+1]\!](h, \eta(l_1))^{\mathtt{L}}) = \rho(\eta(l_1))$ and $\rho([\![h := h+1]\!](h, \eta(l_2))^{\mathtt{L}}) = \rho(\eta(l_2))$, which are clearly the same when $\eta(l_1) = \eta(l_2)$. The major difference between narrow and abstract non-interference is in rules **A5**. In this case we need to consider a narrow assertion for c_2 involving disjunctive domains. This is due to the fact that by definition abstract non-interference checks input properties on singletons while the output of the abstract non-interference assertion for c_1

Table 3. Axiomatic abstract non-interference

A0: $\dfrac{(\eta)[\![c]\!](\mathtt{id}) \sqsubseteq \rho}{(\eta)c(\rho)}$	**A1:** $(\eta)c(\top)$	**A2:** $(\eta)\mathbf{nil}(\rho)$	**A3:** $\dfrac{(\eta)e(\rho),\ x:\mathtt{L}}{(\eta)x := e(\rho)}$
A4: $\dfrac{x:\mathtt{H}}{(\eta)x := e(\rho)}$	**A5:** $\dfrac{(\eta)c_1(\Upsilon(\rho)),\ [\rho]c_2(\Upsilon(\beta))}{(\eta)c_1;c_2(\Upsilon(\beta))}$	**A6:** $\dfrac{\{\rho\}_{\mathtt{L}}\, c\, \{\rho\}_{\mathtt{L}},\ x:\mathtt{H}}{(\rho)\mathbf{while}\ x\ \mathbf{do}\ c\ \mathbf{endw}(\rho)}$	
A7: $\dfrac{(\eta)c(\rho),\ x:\mathtt{L}}{(\eta)\mathbf{while}\ x\ \mathbf{do}\ c\ \mathbf{endw}(\rho)}$	**A8:** $\dfrac{(\eta)c(\rho'),\ \rho' \sqsubseteq \rho}{(\eta)c(\rho)}$	**A9:** $\dfrac{\forall i \in I.(\eta)c(\rho_i)}{(\eta)c(\bigsqcup_{i\in I}\rho_i)}$	**A10:** $\dfrac{\forall i \in I.(\eta)c(\rho_i)}{(\eta)c(\bigsqcap_{i\in I}\rho_i)}$

deals with properties of sets of values. In order to cope with this 'type mismatch", we need to strengthen the natural counterpart of rule **N5** for abstract non-interference. Next example shows that by considering abstract non-interference for c_2 is not sufficient to achieve soundness.

Example 4. Consider Par and the program P in Example 3. We can prove that we have $(\top)l := 4h^2 + 4(Par)$ and $(Par)c(\rho)$, where $\rho \stackrel{\text{def}}{=} Par \cup \{0\}$. But we can show that $\not\models (\top)l := 4h^2 + 4;\ c(\rho)$ since $\rho([\![l := 4h^2+4;\ c]\!](0,\mathbb{Z})^{\mathtt{L}}) = \rho(4) = 2\mathbb{Z}$ while we have $\rho([\![l := 4h^2+4;\ c]\!](1,\mathbb{Z})^{\mathtt{L}}) = \rho(0) = \{0\}$, namely they are different.

Moreover note that **A5** requires that for both c_1 and c_2 the output closures are additive maps, i.e., disjunctive abstract domains, as shown in the following example.

Example 5. Consider the following program P with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$

$$P \stackrel{\text{def}}{=} c_1;c_2 = l := (h \bmod 2)(2l \bmod 4) + (1-(h \bmod 2))(l \bmod 2 + 1);\\ l := (l \bmod 2) * 4h + (1-(l \bmod 2)) * (4h+1)$$

Consider $\rho = \{\mathbb{Z}, 4\mathbb{Z}, 4\mathbb{Z}+1, 4\mathbb{Z}+2, 4\mathbb{Z}+3, \varnothing\}$ (not additive), then $(\top)c_1(\rho)$ since $\forall h \in 2\mathbb{Z}\ \rho([\![c_1]\!](h,\mathbb{Z})^{\mathtt{L}}) = \rho(\{1,2\}) = \mathbb{Z}$, and $\forall h \in 2\mathbb{Z}+1$ we have $\rho([\![c_1]\!](h,\mathbb{Z})^{\mathtt{L}}) = \rho(\{0,2\}) = \mathbb{Z}$. On the other hand it is simple to show that $[\rho]c_2(\rho)$ since this statement leaves unchanged the abstraction of l. But if we consider the composition then we have that $\not\models (\top)P(\rho)$ because if $h \in 2\mathbb{Z}$ then $\rho([\![P]\!](h,\mathbb{Z})^{\mathtt{L}}) = \rho(\{4h, 4h+1\}) = \mathbb{Z}$ while if $h \in 2\mathbb{Z}+1$ then $\rho([\![P]\!](h,\mathbb{Z})^{\mathtt{L}}) = \rho(\{4h+1\}) = 4\mathbb{Z}+1$. Note that the first statement is not secret if we consider the disjunctive completion of ρ in output.

Rule **A6** is equal to **N6**, since also **A5** requires narrow non-interference. Rule **A7** is straightforward from the definition of abstract non-interference and was absent in narrow one for the presence of deceptive flows. The last three rules (**A8**, **A9** and **A10**) change since in abstract non-interference we cannot concretize the input observation. The proof-system in Table 3 is denoted $\mathcal{A} = \mathcal{N} \cup \{\mathbf{A0}, \ldots, \mathbf{A10}\}$ and the proof system without the semantic rules **A0** is denoted as $\mathcal{A}_0 = \mathcal{N}_0 \cup \{\mathbf{A1}, \ldots, \mathbf{A10}\}$. The following theorem proves the soundness of the proof-system $\mathcal{A}_0$ with respect to the standard semantics of IMP.

Theorem 4. *Let* $P \in$ IMP *be a program and* $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$. *If* $\vdash_{\mathcal{A}_0} (\eta)P(\rho)$ *then* $\models (\eta)P(\rho)$.

Example 6. Consider Par and the program P in Example 3. We can prove that $(\top)l := 4h^2+4(Par)$ and $[Par]c(Par)$ therefore $(\top)l := 4h^2+4;\ c(Par)$. Indeed if we consider $l_1 = 4$ and $l_2 = 8$ then clearly $\top(4) = \top(8) = \top$ but $Par([\![l := 4h^2+4;\ c]\!](0,\top)^{\mathtt{L}}) = Par([\![c]\!](0, 4h^2+4)^{\mathtt{L}}) = Par(4h^2+4) = 2\mathbb{Z}$, while $Par([\![l := 4h^2+4;\ c]\!](1,\top)^{\mathtt{L}}) = Par([\![c]\!](1, 4h^2+4)^{\mathtt{L}}) = Par(0) = 2\mathbb{Z}$, namely they are the same.

Example 7. Consider the program fragment

$$P \stackrel{\text{def}}{=} l := 2^h;\ \textbf{while}\ h\ \textbf{do}\ l := 2*l;\ h := h-1\textbf{endw}$$

with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{N}$. First of all we note that $(\top)2^h(\rho_1)$, where $\rho_1 \stackrel{\text{def}}{=} \curlyvee(\{\{2\}^{\mathbb{N}}\} \cup \{\, n \mid n \notin \{2\}^{\mathbb{N}} \,\})$. This means that we can apply **A3**, obtaining $(\top)l := 2^h(\rho_1)$. Consider now the **while**-statement that we denote by c and the closure $\rho_2 \stackrel{\text{def}}{=} \curlyvee(\{\, n\{2\}^{\mathbb{N}} \mid n \in \mathbb{N} \text{ odd} \,\})$. We note that $\{\rho_2\}\ \langle 2*l, l\rangle\ \{\rho_2\}$ and therefore, by **I4** we have $\{\rho_2\}_{\mathtt{L}}\ l := 2*l\ \{\rho_2\}_{\mathtt{L}}$. On the other hand, by **I3** we have $\{\rho_2\}_{\mathtt{L}}\, h := h-1\, \{\rho_2\}_{\mathtt{L}}$, and therefore by **I5** we obtain $\{\rho_2\}_{\mathtt{L}}\, l := 2*l;\ h := h-1\, \{\rho_2\}_{\mathtt{L}}$. Now we can apply **A6** obtaining $[\rho_2]$**while** h **do** $l := 2*l;\ h := h-1$ **endw**(ρ_2) and therefore we use **A5** obtaining $(\top)P(\rho_2)$.

The following example shows that the proof-system $\mathcal{A}_0$ for abstract non interference in Table 3 is not complete.

Example 8. Consider the closure $\rho \stackrel{\text{def}}{=} \{\mathbb{Z}, 2\mathbb{Z}, 4\mathbb{Z}, \varnothing\}$ and consider the program

$$P \stackrel{\text{def}}{=} \textbf{while}\ h\ \textbf{do}\ l := (l \bmod 4) * (l \div 4);\ h := 0\ \textbf{endw}$$

with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$. Note that $(\rho)P(\rho)$ since, for example, $\rho([\![P]\!](1, 2\mathbb{Z})^{\mathtt{L}}) = 2\mathbb{Z} = \rho([\![P]\!](0, 2\mathbb{Z})^{\mathtt{L}})$. But we have that $\not\models \{\rho\}_{\mathtt{L}}\, P\, \{\rho\}_{\mathtt{L}}$ since $\rho([\![P]\!](1,2)^{\mathtt{L}}) = \rho(0) = 4\mathbb{Z} \neq \rho(2) = 2\mathbb{Z}$.

The example above shows that **A6** is not complete, but it is not the only incomplete rule. In particular, by the same argument used in Example 3 for **N5**, **A5** is also incomplete. Even **A7** is incomplete as shown in the following example.

Example 9. Consider $\rho \stackrel{\text{def}}{=} \{\mathbb{Z}, \{0\}, 2\mathbb{Z}_0, 2\mathbb{Z}+1, \varnothing\}$, where $2\mathbb{Z}_0 \stackrel{\text{def}}{=} 2\mathbb{Z} \smallsetminus \{0\}$, and

$$P \stackrel{\text{def}}{=} \textbf{while}\ l_1\ \textbf{do}\ l_2 := \mathtt{iszero}(l_1) * h^2;\ l_1 := 0\ \textbf{endw}$$

with security typing: $t = \langle h : \mathtt{H}, l_1, l_2 : \mathtt{L}\rangle$ and $\mathtt{iszero}(x) = 1$ if $x = 0$ and $\mathtt{iszero}(x) = 0$ otherwise. Then we show that $\not\models (\rho)l_2 := \mathtt{iszero}(l_1) * h^2;\ l_1 := 0(\rho)$ since, if we take the low input $\langle 0, 2\mathbb{Z}_0\rangle$ then we have $\rho([\![l_2 := \mathtt{iszero}(l_1) * h^2;\ l_1 := 0]\!](1, \langle 0, 2\mathbb{Z}_0\rangle)^{\mathtt{L}}) = \rho(\langle 0, 1\rangle) = \langle 0, 2\mathbb{Z}+1\rangle \neq \langle 0, 2\mathbb{Z}_0\rangle = \rho([\![l_2 := \mathtt{iszero}(l_1) * h^2;\ l_1 := 0]\!](2, \langle 0, 2\mathbb{Z}_0\rangle)^{\mathtt{L}})$. But it is worth noting that $(\rho)P(\rho)$ since for example $\rho([\![P]\!](h, \langle 0, 2\mathbb{Z}_0\rangle)^{\mathtt{L}}) = \langle 0, 2\mathbb{Z}_0\rangle$ and $\rho([\![P]\!](h, \langle 2\mathbb{Z}_0, 2\mathbb{Z}_0\rangle)^{\mathtt{L}}) = \langle 0, 0\rangle$.

All the other rules are complete. As above, for the proof-system for narrow abstract non-interference $\mathcal{N}$, also for abstract non-interference, the semantic rule **A0** makes $\mathcal{A}$ complete. This is a straight consequence of Th. 1.

Corollary 2. *The proof-system $\mathcal{A}$ is complete.*

Next result specifies a relation between derivations in the narrow and abstract non-interference proof systems. This result is in accordance with the expected relation between narrow and abstract non-interference, the first being stronger.

Theorem 5. *Let $P \in$ IMP be a program and $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$. If $\vdash_{\mathcal{N}_0} [\eta]P(\rho)$ then $\vdash_{\mathcal{A}_0} (\eta)P(\rho)$.*

Next example shows that $\mathcal{A}$ is strictly weaker than $\mathcal{N}$. We show that if $\models [\eta]P(\rho)$ and $\vdash_{\mathcal{A}_0} (\eta)P(\rho)$, the fact that $[\eta]P(\rho) \Rightarrow (\eta)P(\rho)$ does not imply that $\vdash_{\mathcal{N}_0} [\eta]P(\rho)$.

Example 10. Consider the property Par and the program: $P \stackrel{\text{def}}{=} h := h+1;\ l := 2*h$, with security typing: $t = \langle h : \mathtt{H}, l : \mathtt{L}\rangle$ and $\mathbb{V}^{\mathtt{H}} = \mathbb{V}^{\mathtt{L}} = \mathbb{Z}$. Note that $[Sign]P(Par)$ since $\forall l \in \mathbb{V}^{\mathtt{L}},\ h \in \mathbb{V}^{\mathtt{H}}$ we have $Par([\![P]\!](h,l)^{\mathtt{L}}) = Par(2*h) = 2\mathbb{Z}$. This means also that $\models (Sign)P(Par)$. But note that $\not\models [Sign]h := h+1(Par)$ since $Sign(2) = Sign(3) = \mathbb{Z}^+$ and $Par([\![h := h+1]\!](h,2)^{\mathtt{L}}) = Par(2) = 2\mathbb{Z} \neq Par([\![h := h+1]\!](h,3)^{\mathtt{L}}) = Par(3) = 2\mathbb{Z}+1$. This means that $\not\vdash_{\mathcal{N}_0} [Sign]P(Par)$. On the other hand we have that $\vdash_{\mathcal{A}_0} (Sign)h := h+1(Par)$ and $\vdash_{\mathcal{N}_0} [Par]l := 2*h(Par)$, therefore we can use **A5** since Par is disjunctive, and therefore we infer $(Sign)P(Par)$.

5 Discussion

We have introduced a sound proof-system for both narrow and abstract non-interference. The advantage of a proof-system for abstract non-interference is that checking abstract non-interference can be easily mechanized. Both $\mathcal{N}$ and $\mathcal{A}$ can benefit of standard abstract interpretation methods for generating basic certificates for simple program fragments (rules **N0** and **A0**). The other rules allow us to combine certificates from program fragments in a proof-theoretic derivation of harmless models of attackers, certifying program secrecy. The interest in this technology is mostly related with its use in *a la* proof carrying code (PCC) verification of abstract non-interference, when mobile code is allowed. In this case in a PCC architecture, the code producer may create an abstract non-interference certificate that attests to the fact that the code secrecy cannot be violated by the corresponding model of the attacker. Then the code consumer may validate the certificate to check that the foreign code is secure for the corresponding model of attacker. The implementation of this technology requires an appropriate choice of a logic for specifying abstractions and an adequate logical framework where the logic can be manipulated. We believe that predicate abstraction [15, 21] is a fairly simple and easily mechanizable way for reasoning about abstract domains. More appropriate logics can be designed following the ideas in [2], even though a mechanizable logic for reasoning about abstractions is currently a major challenge in this field and deserves further investigations. The language we used is quite simple. Even though abstract non-interference made secrecy a purely semantics problem, any extension of IMP and its semantics with for example probabilistic choice, non terminating computations, and concurrency, may require a redesign of the proof-systems for narrow and abstract non-interference. It would be particularly interesting to extend IMP with concurrency. The

main interest in this extension deals both with the chance to reduce protocol verification to non-interference problems and with the possibility of modeling active attackers as abstract interpretations in language-based security. The models of attackers developed in abstract non-interference are indeed passive [16]. Active attackers would be particularly relevant in order to extend abstract non-interference as a language-based tool for protocol validation.

Acknowledgments. This work is supported by the Italian-MIUR Projects *COFIN02-CoVer: Constraint-based verification of reactive systems* and *FIRB: Abstract interpretation and model checking for the verification of embedded systems*. We also thank the anonymous referees for their helpful comments to the previous versions of this paper.

References

1. G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, 1980.
2. A. Appel. Foundational proof-carrying code. In *Proc. of the 16th IEEE Symp. on Logic in Computer Science* (*LICS '01*), pages 247–258. IEEE Computer Society Press, Los Alamitos, Calif., 2001.
3. K. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, Berlin, 1997.
4. D. E. Bell and E. Burke. A software validation technique for certification: The methodology. Technical Report MTR-2932, MITRE Corp. Badford, MA, 1974.
5. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp. Badford, MA, 1973.
6. D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002.
7. E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating System Review*, 11(5):133–139, 1977.
8. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47,103, 2002.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages* (*POPL '77*), pages 238–252. ACM Press, New York, 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, New York, 1979.
11. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. of Conf. Record of the 19th ACM Symp. on Principles of Programming Languages* (*POPL '92*), pages 83–94. ACM Press, New York, 1992.
12. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
13. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
14. D. E. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

15. C. Flanagan and S. Qadeer. Pedicate abstraction for software verification. In *Proc. of Conf. Record of the 29th ACM Symp. on Principles of Programming Languages* (*POPL '02*), pages 191–202. ACM Press, New York, 2002.
16. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM-Press, NY, 2004.
17. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. of The 8th Internat. Static Analysis Symp. (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer-Verlag, 2001.
18. R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program*, 32(1-3):177–210, 1998.
19. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
20. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86. IEEE Computer Society Press, 1984.
21. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Internat. Conf. on Computer Aided Verification* (*CAV '97*), volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, Berlin, 1997.
22. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37:113–138, 2000.
23. P. Laud. Semantics and program analysis of computationally secure information flow. In *In Programming Languages and Systems, 10th European Symp. On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
24. G. Necula. Proof-carrying code. In *Proc. of Conf. Record of the 24th ACM Symp. on Principles of Programming Languages* (*POPL '97*), pages 106–119. ACM Press, New York, 1997.
25. F. Ranzato and F. Tapparo. Making abstract model checking strongly preserving. In M. Hermenegildo and G. Puebla, editors, *Proc. of The 9th Internat. Static Analysis Symp. (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 411–427. Springer-Verlag, 2002.
26. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1):5–19, 2003.
27. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
28. C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP'00*, pages 254–267. ACM Press, New York, 2000.
29. D. Volpano. Safety versus secrecy. In *Proc. of the 6th Static Analysis Symp. (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311. Springer-Verlag, 1999.
30. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
31. G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.
32. M. Zanotti. Security typings by abstract interpretation. In M. Hermenegildo and H. Puebla, editors, *Proc. of The 9th Internat. Static Analysis Symp. (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 360–375. Springer-Verlag, 2002.