Statically Analyzing Information Flows

An Abstract Interpretation-based Hyperanalysis for Non-Interference

Isabella Mastroeni
University of Verona - Dept. of Computer Science
Verona, Italy
isabella.mastroeni@univr.it

Michele Pasqua*
University of Verona - Dept. of Computer Science
Verona, Italy
michele.pasqua@univr.it

ABSTRACT

In the context of systems security, information flows play a central role. Unhandled information flows potentially leave the door open to very dangerous types of attacks, such as code injection or sensitive information leakage. Information flows verification is based on the definition of Non-Interference [8], which is known to be an hyperproperty [7], i.e., a property of sets of executions. The sound verification of hyperproperties is not trivial [3, 16]: It is not easy to adapt classic verification methods, used for trace properties, in order to deal with hyperproperties. In the present work, we design an abstract interpretation-based static analyzer soundly checking Non-Interference. In particular, we define an hyper abstract domain, able to approximate the information flows occurring in the analyzed programs.

CCS CONCEPTS

• **Software and its engineering** → **Software verification**; • **Security and privacy** → *Information flow control*;

KEYWORDS

Static analysis, Information flows, Abstract interpretation, Hyperproperties verification, Non-Interference

ACM Reference Format:
Isabella Mastroeni and Michele Pasqua. 2019. Statically Analyzing Information Flows: An Abstract Interpretation-based Hyperanalysis for Non-Interference. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3297498

1 INTRODUCTION

Programs security often relies on how information is propagated during programs executions. Historically, in the context of confidentiality, *access control* has been the main means of preventing information from being disseminated. As the name indicates, access control verifies the program rights at entry-point. However, it is inadequate in many situations, in fact the program may leak/compromise information after the access check. Instead *information flow control* tracks how information propagates through the program during execution to make sure that it handles the information safely. A security mechanism, given a program and a specification stating which flows of information are allowed and which are not, checks whether the program is secure w.r.t. the given specification. In general, unhandled information flows potentially leave the door open to very dangerous types of attacks, such as code injection or sensitive information leakage.

Information flow control is based on the definition of Non-Interference [8], where variables are marked as public or private, and flows from the public ones to the private ones are forbidden. Indeed, a program is said to be *non-interferent* if the values of the public outputs (i.e., at the end of the execution) do not depend on the values of the private inputs (i.e., at the beginning of the execution), in other words if whenever we start from states agreeing on the public inputs, the public observable outputs are the same.

The problem of Non-Interference verification is far from solved. Indeed, Non-Interference is a hard problem to solve due to two main reasons: it is undecidable and it is not even a trace property, in the sense that it cannot be verified on single executions, since it requires the comparison of multiple computations. In other words it is an undecidable *hyperproperty* [7].

Both these problems are tackled by approximation, in particular by designing *trace properties* stronger than Non-Interference, but which can be verified by means of standard techniques, inherited from classic programs verification for trace properties. Unfortunately, this means that we introduce two degrees of approximation, one for approximating Non-Interference as a trace property and one for making it decidable.

Whilst for trace properties standard analysis techniques can be used with an acceptable degree of approximation, the verification, and hence the approximation, of hyperproperties is a hard problem to solve in a sufficiently precise way. Being a hyperproperty means to be modeled as a set of sets, and the extra level of sets introduces a lot of technical problems for approximation-based verification methods. Hence, in order to obtain significant results w.r.t. analysis precision, we need new verification methods which approximate sets of sets (instead of just sets) of executions [16].

In [17], the authors propose a general framework for designing abstract interpretations for hyperproperties verification. In the present work, our aim is to instantiate this framework to the verification of Non-Interference, allowing us to design a prototype analyzer of Non-Interference for a toy programming language. In particular, we first design the abstract domain denoting the hyperproperty that has to be satisfied by the semantics of non-interfering programs, i.e., the values of public variables must be always the same single

*Corresponding author.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297498

value. Unfortunately, this ideal domain (observing the precise single public values of all the computations starting from the same public input) is infinite and does not guarantee decidability of the analysis, hence we need to introduce further abstractions. Once we fix the domain of the abstract observable hyperproperty we define the abstract hypersemantics computing a program's executions on the abstract domain, exactly as it happens in classic static analysis, but with the only difference that we are abstracting a hypersemantics into a hyperdomain. To the best of our knowledge, this is the first attempt to provide a sound analyzer of Non-Interference exploiting the expressiveness of hyperproperties.

Finally, in order to test the feasibility of the proposed approach we have implemented a prototype (called nonInterfer) of the designed analyzer. This prototype exhibits a good trade-off between verification speed and precision. It can be tested at the link http://bit.do/noninterfer, via a web interface.

2 BACKGROUND

In this section, we set the background concepts needed to introduce our prototype analyzer. First, we define a toy programming language for which we design the analyzer. Then we make a brief introduction to hyperproperties and, finally, we have two little paragraphs, one for abstract interpretation and one for Non-Interference.

2.1 Programs

Programs are written in a simple deterministic imperative language Imp, whose grammar is the following:

$$\begin{aligned}
&\text{Aexp} \ni a ::= x \mid n \mid a \oplus a \mid (a)\\
&\text{Bexp} \ni b ::= \mathtt{tt} \mid \mathtt{ff} \mid b \wedge b \mid b \vee b \mid \neg b \mid a \bowtie a \mid (b)\\
&\text{Imp} \ni P ::= \mathtt{skip} \mid x := a \mid P\ ;\ P \mid \mathtt{if}\ b\ \{P\}\ \mathtt{else}\ \{P\} \mid \mathtt{while}\ b\ \{P\}\\
&\qquad\qquad \text{with } \oplus \in \{+, -, *\} \text{ and } \bowtie \in \{=, \neq, <, \leq\}
\end{aligned}$$

Variables $x \in \mathsf{Var}$ range over integer values, hence arithmetic expressions evaluate to values in $\mathbb{Z}$. Boolean expressions evaluate to boolean values in $\mathbb{B} \triangleq \{\mathbf{tt}, \mathbf{ff}\}$. The semantics of the language is given on top of *memories*, namely maps from variables to values. Let $\mathsf{Mem} \triangleq \mathsf{Var} \to \mathbb{Z}$ be the set of programs memories. We define the semantics of a program inductively from its syntax. In particular, it is built on top of the small-step operational semantics (SOS) of Imp. This latter models the execution of programs step by step and it is specified by a set of inference rules modifying *configurations*. A configuration $\langle m, P\rangle \in \mathsf{Mem} \times \mathsf{Imp}$ represents the current memory m in which the program P has to be executed. The SOS inference rules in Fig. 1 describe how configurations evolve during time, until a final configuration of the form $\langle m, \mathtt{skip}\rangle$ is reached (if ever). The SOS rules rely on the semantics for arithmetic expressions $(\!|a|\!) \in \mathsf{Mem} \to \mathbb{Z}$ and for boolean expressions $(\!|b|\!) \in \mathsf{Mem} \to \mathbb{B}$. These latter are big-step semantics, since we are not interested in the intermediate steps of computation for expressions. In Fig. 2 we have the definition of the (big-step) semantics for expressions. We denote with $\rightarrowtriangle^k$ the application of $\rightarrowtriangle$ a number $k > 1$ of times, hence $\langle m, P\rangle \rightarrowtriangle^k \langle m', P'\rangle$ if there exists a sequence σ of k configurations such that: $\sigma_0 = \langle m, P\rangle$, $\sigma_{k-1} = \langle m', P'\rangle$ and $\forall i \in [0, k-1)\,.\,\sigma_i \rightarrowtriangle \sigma_{i+1}$. We denote with $\langle m, P\rangle \rightarrowtriangle^* \langle m', P'\rangle$ the fact that there exists a $k > 1$ such that $\langle m, P\rangle \rightarrowtriangle^k \langle m', P'\rangle$.

A program P, starting in the memory m, terminates, yielding the memory m', iff $\langle m, P\rangle \rightarrowtriangle^* \langle m', \mathtt{skip}\rangle$. Conversely, P diverges (on m) iff it is possible to apply $\rightarrowtriangle$ to $\langle m, P\rangle$ infinitely many times.

2.2 Hyperproperties

In the field of information security, *verification* is the general process of checking if a system complies with a specification, i.e., a formal description of what systems are allowed and/or are not allowed to do. The majority of works about verification deal with particular specifications, those expressible with the so called *trace properties*, often simply called "properties". They are defined in terms of single executions and hence cannot express specifications which need to take into account relations between executions. In [7], *hyperproperties* were introduced in order to formalize those specifications which are not trace properties. When systems are programs in Imp then program executions can be modeled as *traces* of state denotations. Examples of traces could be the set of all possible sequences of memories generated by the SOS of programs or the set of reachable memories, etc. The behavior (semantics) $\mathcal{S}^P$ of a system (program) P is modeled as a set of traces, one for each possible input. In this setting, trace properties are sets of traces and hyperproperties are sets of sets of traces. The satisfiability relation is the set inclusion for trace property and the set membership for hyperproperties. Namely, P satisfies the trace property $\mathfrak{P}$, written $P \models \mathfrak{P}$, iff $\mathcal{S}^P \subseteq \mathfrak{P}$ and it satisfies the hyperproperty $\mathfrak{Hp}$, written $P \models \mathfrak{Hp}$, iff $\mathcal{S}^P \in \mathfrak{Hp}$. The satisfiability for hyperproperties can be restated using set inclusion, in fact $\mathcal{S}^P \in \mathfrak{Hp}$ iff $\{\mathcal{S}^P\} \subseteq \mathfrak{Hp}$. The strongest program trace property of P is precisely $\mathcal{S}^P$ and its strongest hyperproperty is $\{\mathcal{S}^P\}$ [10, 16]. Strongest here means that every property[hyperproperty] satisfied by P is implied by (i.e., contains) $\mathcal{S}^P[\{\mathcal{S}^P\}]$. In order to prove that a program P satisfies

$$\frac{\langle m, P_1\rangle \rightarrowtriangle \langle m', P_3\rangle}{\langle m, P_1\ ;\ P_2\rangle \rightarrowtriangle \langle m', P_3\ ;\ P_2\rangle} \qquad \frac{-}{\langle m, \mathtt{skip}\ ;\ P\rangle \rightarrowtriangle \langle m, P\rangle}$$

$$\frac{(\!|b|\!)m = \mathbf{tt}}{\langle m, \mathtt{if}\ b\ \{P_1\}\ \mathtt{else}\ \{P_2\}\rangle \rightarrowtriangle \langle m, P_1\rangle} \qquad \frac{(\!|b|\!)m = \mathbf{ff}}{\langle m, \mathtt{if}\ b\ \{P_1\}\ \mathtt{else}\ \{P_2\}\rangle \rightarrowtriangle \langle m, P_2\rangle}$$

$$\frac{(\!|b|\!)m = \mathbf{tt}}{\langle m, \mathtt{while}\ b\ \{P\}\rangle \rightarrowtriangle \langle m, P\ ;\ \mathtt{while}\ b\ \{P\}\rangle} \qquad \frac{(\!|b|\!)m = \mathbf{ff}}{\langle m, \mathtt{while}\ b\ \{P\}\rangle \rightarrowtriangle \langle m, \mathtt{skip}\rangle}$$

$$\frac{(\!|a|\!)m = n}{\langle m, x := a\rangle \rightarrowtriangle \langle m[x \leftarrowtail n], \mathtt{skip}\rangle}$$

Figure 1: Small-step operational semantics of Imp.

Arithmetic expressions: $(\!|a|\!) \in \mathsf{Mem} \to \mathbb{Z}$

$$(\!|n|\!)m = n \qquad (\!|x|\!)m = m(x) \qquad (\!|(a)|\!)m = (\!|a|\!)m$$
$$(\!|a_1 \oplus a_2|\!)m = (\!|a_1|\!)m \oplus (\!|a_2|\!)m \quad \text{where } \oplus \in \{+, -, *\}$$

Boolean expressions: $(\!|b|\!) \in \mathsf{Mem} \to \mathbb{B}$

$$(\!|\mathtt{tt}|\!)m = \mathbf{tt} \qquad (\!|\mathtt{ff}|\!)m = \mathbf{ff} \qquad (\!|(b)|\!)m = (\!|b|\!)m \qquad (\!|\neg b|\!)m = \neg(\!|b|\!)m$$
$$(\!|b_1 \wedge b_2|\!)m = (\!|b_1|\!)m \wedge (\!|b_2|\!)m \qquad (\!|b_1 \vee b_2|\!)m = (\!|b_1|\!)m \vee (\!|b_2|\!)m$$
$$(\!|a_1 \bowtie a_2|\!)m = (\!|a_1|\!)m \bowtie (\!|a_2|\!)m \quad \text{where } \bowtie \in \{=, \neq, <, \leq\}$$

Figure 2: Big-step semantics for expressions.

a property[hyperproperty] we use an over-approximation of the strongest P property[hyperproperty].

Some specifications can be expressed as trace properties, like Access Control, and others cannot, like Non-Interference. Among all hyperproperties there are some easier to verify. In [17], the authors define k-bounded subset-closed hyperproprties: these specifications can be refuted just exhibiting a counterexample set consisting in at most k traces. It turns out that several interesting hyperproperties are k-bounded (like Non-Interference). In Sec. 4 we will see how we can exploit this fact in order to verify Non-Interference.

2.3 Abstract Interpretation

The (concrete) semantic of a program, namely its behavior, is a representation of all its possible executions by means of a set of mathematical objects. This set is, in general, not computable: It is not possible for a program to represent and to compute all possible executions of any program starting from all its possible inputs. Clearly all non trivial properties of the concrete semantics of a program are undecidable: It is not possible for a program to answer any question about the executions of any program. Abstract interpretation is born, as a theory for soundly approximating the semantics of discrete dynamic systems. The approximation consists in the observation of the semantics at a specified level of abstraction, focusing only on some important aspects of computations. In this setting, abstract interpretation allows us to compute an abstract semantics of the program, depending on the properties of interest. The approximation is sound by design, in the sense that what holds in the abstract holds also in the concrete (no false negative).

One of the fundamental aspects of abstract interpretation is that the majority of the features of the approximation process are specified only by the (abstract) domain of mathematical objects chosen for representing the properties of interest. A theory of domains for abstract interpretation was defined in [9], based on the notion of *Galois insertion*. A Galois insertion (C, α, γ, A) consists of two partially ordered sets $\langle C, \leq_C\rangle$, $\langle A, \leq_A\rangle$ and two monotone functions $\alpha \in C \to A$, $\gamma \in A \to C$ such that for all c in C and a in A it holds: $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ and $\alpha \circ \gamma = id$ (the identity function $\lambda x\,.\,x$). C is the concrete domain, A is the abstract domain, α is the abstraction function and γ is the concretization function. Sometimes, abstract interpretations are given by means of Galois connections (instead of Galois insertions), relaxing the constraints $\alpha\circ\gamma = id$. This do not restrict the generality of the framework, since every connection can be transformed in an insertion, eliminating the redundant elements in the abstract domain.

Let $f \in C \to C$ be a function on the concrete domain and $f^\sharp \in A \to A$ be a function on the abstract domain. $f^\sharp$ is a sound (or correct) approximation of f if $f \circ \gamma = \gamma \circ f^\sharp$ or, equivalently, if $\alpha \circ f = f^\sharp \circ \alpha$ [9]. The *best correct approximation* of f in A, defined as $f^{\text{bca}} \triangleq \alpha \circ f \circ \gamma$, is sound by construction [9]. Hence, in order to prove that a given abstract function $f^\sharp$ is sound, it is sufficient to prove that it approximates f^{bca}, i.e., if $\forall a \in A\,.\, f^{\text{bca}}(a) \leq_A f^\sharp(a)$.

2.4 Non-Interference

Computing system security relies on how information is propagated during system execution. Information flow control tracks how information propagates through the system during execution to make sure that the system handles the information securely. In the last forty years, a lot of specifications for information flow control have been proposed (see [14] for a survey). They differ on the formal definitions but share the following common informal understanding of confidentiality: the lack of dependencies on confidential information. This is precisely the absence of strong dependency of [8] and it is the base of confidentiality information flow specifications. The idea of [8] is that there exists an information flow from variable x to variable y in a program P whenever variety in x is conveyed to y by the execution of P. The original formulation of Non-Interference [8] takes in consideration only two security levels: *private* (H), i.e., information that has to be kept secret, and *public* (L), i.e., information that could be freely released. A program is said non-interferent if there are no information flows from private to public variables, and interferent otherwise.

3 SIMPLIFYING NON-INTERFERENCE VERIFICATION

In the following, we define Non-Interference for programs in Imp and we show how to derive a verification mechanism for it. The classic notion of Non-Interference [8] checks the input/output relation between executions, so we represent programs computations with just the initial and the final memories. Furthermore, it is not termination sensitive, hence we ignore divergent computations. With these premises, the set of executions denotations is $\mathsf{Mem} \times \mathsf{Mem}$. It is well known that information flows are hyperproperties [7], indeed Non-Interference $\mathtt{NI} \in \wp(\wp(\mathsf{Mem} \times \mathsf{Mem}))$ is defined as:

$$\mathtt{NI} \triangleq \left\{ X \subseteq \mathsf{Mem} \times \mathsf{Mem} \;\middle|\; \begin{array}{l} \forall \langle \mathsf{m}_1, \mathsf{m}_1'\rangle, \langle \mathsf{m}_2, \mathsf{m}_2'\rangle \in X\,. \\ \mathsf{m}_1 =_{\mathsf{L}} \mathsf{m}_2 \Rightarrow \mathsf{m}_1' =_{\mathsf{L}} \mathsf{m}_2' \end{array} \right\}$$

The relation $=_{\mathsf{L}}$ says that memories agree on public variables, i.e., $\mathsf{m} =_{\mathsf{L}} \mathsf{m}'$ iff $\forall \mathsf{x}\,.\,\Gamma(\mathsf{x}) = \mathsf{L} \Rightarrow \mathsf{m}(\mathsf{x}) = \mathsf{m}'(\mathsf{x})$. Here $\Gamma \in \mathsf{Var} \to \{\mathsf{L}, \mathsf{H}\}$ is the typing environment, assigning variables to security levels.

Given a program $\mathsf{P} \in \mathsf{Imp}$, its semantics (i.e., its strongest property) on $\mathsf{Mem} \times \mathsf{Mem}$ is ${}^{\times}\mathcal{S}^{\mathsf{P}} \in \wp(\mathsf{Mem}^{\mathsf{P}} \times \mathsf{Mem}^{\mathsf{P}})$, defined as ${}^{\times}\mathcal{S}^{\mathsf{P}} \triangleq \{\langle \mathsf{m}, \mathsf{m}'\rangle \mid \langle \mathsf{m}, \mathsf{P}\rangle \rightarrowtriangle^* \langle \mathsf{m}', \mathsf{skip}\rangle\}$. Its hypersemantics (i.e., its strongest hyperproperty) is ${}^{\times}\mathcal{HS}^{\mathsf{P}} \triangleq \{{}^{\times}\mathcal{S}^{\mathsf{P}}\}$. Hence we have that $\mathsf{P} \models \mathtt{NI}$ iff ${}^{\times}\mathcal{S}^{\mathsf{P}} \in \mathtt{NI}$ or, equivalently, iff ${}^{\times}\mathcal{HS}^{\mathsf{P}} \subseteq \mathtt{NI}$.

Now we will see how the verification process for Non-Interference can be made simpler. Since NI is a 2-bounded subset-closed hyperproperty [17], we have that $\mathsf{P} \models \mathtt{NI}$ iff ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2} \triangleq \{X \subseteq {}^{\times}\mathcal{S}^{\mathsf{P}} \mid |X| = 2\} \subseteq \mathtt{NI}$. We can note that ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}$ can be partitioned in

$${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{\neq} \triangleq \left\{\{\langle \mathsf{m}_1, \mathsf{m}_1'\rangle, \langle \mathsf{m}_2, \mathsf{m}_2'\rangle\} \;\middle|\; \mathsf{m}_1 \neq_{\mathsf{L}} \mathsf{m}_2\right\} \text{ and}$$

$${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=} \triangleq \left\{\{\langle \mathsf{m}_1, \mathsf{m}_1'\rangle, \langle \mathsf{m}_2, \mathsf{m}_2'\rangle\} \;\middle|\; \mathsf{m}_1 =_{\mathsf{L}} \mathsf{m}_2\right\}$$

namely, ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{\neq} \cup {}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=} = {}^{\times}\mathcal{HS}^{\mathsf{P}\#2}$ and ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{\neq} \cap {}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=} = \varnothing$. At this point, as far as NI is concerned, we can observe that ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{\neq} \subseteq \mathtt{NI}$ always holds, hence we have the following result.

Proposition 3.1. $\mathsf{P} \models \mathtt{NI}$ *iff* ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=} \subseteq \mathtt{NI}$.

Example 3.2. Consider $\mathsf{P} = \mathsf{x} := \mathsf{y}\,;\, \mathsf{y} := 1$, with $\Gamma(\mathsf{x}) = \mathsf{L}$ and $\Gamma(\mathsf{y}) = \mathsf{H}$. Suppose that variables can only take values 0 and 1. In order to simplify, consider the following encoding of memories: $\mathsf{a} \triangleq [\mathsf{x}\mapsto 0\ \mathsf{y}\mapsto 0]$, $\mathsf{b} \triangleq [\mathsf{x}\mapsto 0\ \mathsf{y}\mapsto 1]$, $\mathsf{c} \triangleq [\mathsf{x}\mapsto 1\ \mathsf{y}\mapsto 0]$, $\mathsf{d} \triangleq [\mathsf{x}\mapsto 1\ \mathsf{y}\mapsto 1]$. The semantics of P is ${}^{\times}\mathcal{S}^{\mathsf{P}} = \{\langle \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{b}, \mathsf{d}\rangle, \langle \mathsf{c}, \mathsf{b}\rangle, \langle \mathsf{d}, \mathsf{d}\rangle\}$. Then ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=} = \{\{\langle \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{b}, \mathsf{d}\rangle\}, \{\langle \mathsf{c}, \mathsf{b}\rangle, \langle \mathsf{d}, \mathsf{d}\rangle\}\}$ and ${}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{\neq} = \{\{\langle \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{c}, \mathsf{b}\rangle\}, \{\langle \mathsf{a}, \mathsf{b}\rangle, \langle \mathsf{d}, \mathsf{d}\rangle\}, \{\langle \mathsf{b}, \mathsf{d}\rangle, \langle \mathsf{c}, \mathsf{b}\rangle\}, \{\langle \mathsf{b}, \mathsf{d}\rangle, \langle \mathsf{d}, \mathsf{d}\rangle\}\}$. ■

The verification process for Non-Interference can be simplified even further. Take in consideration the following approximation $\alpha_r \in \wp(\wp(\mathsf{Mem} \times \mathsf{Mem})) \to \wp(\wp(\mathsf{Mem}) \times \wp(\mathsf{Mem}))$:

$$\alpha_r \triangleq \lambda \mathcal{X} \,.\, \left\{ \langle \{a \mid \langle a, b\rangle \in X\}, \{b \mid \langle a, b\rangle \in X\}\rangle \;\middle|\; X \in \mathcal{X} \right\}$$

which abstracts the relation between the input and the output memories of traces (i.e., we only keep the relation between the whole *set* of input memories and the whole *set* of output memories). Then consider a further approximation $\alpha_n \in \wp(\wp(\mathsf{Mem}) \times \wp(\mathsf{Mem})) \to \wp(\wp(\mathsf{Mem})) \times \wp(\wp(\mathsf{Mem}))$:

$$\alpha_n \triangleq \lambda X \,.\, \langle \{A \mid \langle A, B\rangle \in X\}, \{B \mid \langle A, B\rangle \in X\}\rangle$$

which abstracts the relation between sets of traces (i.e., we only keep the relation between the set of all the possible input sets of memories and the set of all the output sets of memories). Finally, setting $\alpha_{nr} \triangleq \alpha_n \circ \alpha_r$ we have that α_{nr} and its left adjoint α_{nr}^- form the Galois connection[1]:

$$\langle \wp(\wp(\mathsf{Mem} \times \mathsf{Mem})), \subseteq \rangle \underset{\alpha_{nr}}{\overset{\alpha_{nr}^-}{\leftrightarrows}} \langle \wp(\wp(\mathsf{Mem})) \times \wp(\wp(\mathsf{Mem})), \subseteq \rangle$$

Now we are able to state a simplified form of verification check for Non-Interference:

$$\mathsf{P} \models \mathtt{NI} \Leftrightarrow \alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2}) \subseteq \alpha_{nr}(\mathtt{NI})$$

We can note that $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2}) \subseteq \alpha_{nr}(\mathtt{NI})$ iff the set $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2})_{\dashv}$ contains only sets of memories $\{\mathsf{m}, \mathsf{m}'\}$ agreeing on the L variables, i.e., such that $\mathsf{m} =_{\mathsf{L}} \mathsf{m}'$. Here, given a pair $\langle x, y\rangle$, we denote with $\langle x, y\rangle_{\dashv} \triangleq y$ its projection on the second element (analogously, we denote with $\langle x, y\rangle_{\vdash} \triangleq x$ its projection on the first element).

Example 3.3. Continuing Ex. 3.2, we have that $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2})$ is equal to $\alpha_n(\{\langle\{\mathsf{a},\mathsf{b}\},\{\mathsf{b},\mathsf{d}\}\rangle, \langle\{\mathsf{c},\mathsf{d}\},\{\mathsf{b},\mathsf{d}\}\rangle\})$, which is the pair $\langle\{\{\mathsf{a},\mathsf{b}\},\{\mathsf{c},\mathsf{d}\}\},\{\{\mathsf{b},\mathsf{d}\}\}\rangle$. In this case, program P does not satisfy Non-Interference since $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2})_{\dashv} = \{\{\mathsf{b},\mathsf{d}\}\}$ and $\mathsf{b} \neq_{\mathsf{L}} \mathsf{d}$. ∎

Basically, we can verify Non-Interference just checking whether $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2})_{\dashv}$ satisfies a hyperproperty on the set of execution denotations Mem, as state by the following proposition.

Proposition 3.4. $\mathsf{P} \models \mathtt{NI}$ *iff* $\alpha_{nr}({}^{\times}\mathcal{HS}_{=}^{\mathsf{P}\#2})_{\dashv} \subseteq \mathtt{equiv}^{\mathsf{L}}$, *where* $\mathtt{equiv}^{\mathsf{L}} \triangleq \left\{ X \subseteq \mathsf{Mem} \;\middle|\; \forall \mathsf{m}, \mathsf{m}' \in X \,.\, \mathsf{m} =_{\mathsf{L}} \mathsf{m}' \right\}$.

This simplifies a lot the verification process for Non-Interference: We can build a hypersemantics computing on $\wp(\wp(\mathsf{Mem}))$ instead of $\wp(\wp(\mathsf{Mem} \times \mathsf{Mem}))$.

4 ABSTRACT HYPERSEMANTICS

Classic methods for the verification of trace properties rely on the so called collecting semantics, namely a representation of a program's behaviors starting from every possible input. This semantics essentially computes the strongest program property. With a computable approximation of this latter the verification process is made feasible. Unfortunately, for hyperproperties this is not sufficient anymore: a collecting hypersemantics, computing the strongest program hyperproperty, is needed [3, 16]. Then the verification is performed with an approximation of this latter. In this section we give a collecting hypersemantics and its abstract version for the verification of Non-Interference. In particular, we follow the construction introduced in [17] for k-bounded subset-closed hyperproperties (indeed Non-Interference is 2-bounded). First, we give the definition of the collecting hypersemantics, computing at the level of sets of sets. Then we instantiate the hyperlevel constants domain of [17] to Non-Interference. This latter, in its original formulation, is not machine-representable [11], namely it has an uncountable set of elements. Moreover, it contains infinite ascending chains (i.e., it is not ACC), inducing potential computation divergence. Hence, we approximate it in order to make its implementation feasible. Finally, we give the definition of the abstract semantics.

4.1 Collecting Hypersemantics

As we have seen in Sec. 3 (Prop. 3.4), we can verify Non-Interference in a simpler domain, namely we can collect memories instead of input/output traces of memories. We move from $\wp(\mathsf{Mem} \times \mathsf{Mem})$ to $\wp(\mathsf{Mem})$, with the abstraction $\alpha_{\mathcal{R}} \triangleq \lambda X \,.\, \{\mathsf{m}' \mid \langle \mathsf{m}, \mathsf{m}'\rangle \in X\}$ collecting all final memories of input/output traces. Hence, we have that the strongest trace property of P induced by its semantics on the domain Mem is ${}^{\mathcal{R}}\mathcal{S}^{\mathsf{P}} \triangleq \alpha_{\mathcal{R}}({}^{\times}\mathcal{S}^{\mathsf{P}}) = \{\mathsf{m}' \mid \exists \mathsf{m} \,.\, \langle \mathsf{m}, \mathsf{P}\rangle \twoheadrightarrow^{*} \langle \mathsf{m}', \mathsf{skip}\rangle\}$. Analogously, the strongest hyperproperty of P on Mem is ${}^{\mathcal{R}}\mathcal{HS}^{\mathsf{P}} \triangleq \{\alpha_{\mathcal{R}}({}^{\times}\mathcal{S}^{\mathsf{P}})\} = \{{}^{\mathcal{R}}\mathcal{S}^{\mathsf{P}}\}$.

In order to compute this semantics, we follow the construction given in [17] (Sect. 3.1). We start with a classic collecting semantics $[\![\mathsf{P}]\!] \in \wp(\mathsf{Mem}) \to \wp(\mathsf{Mem})$, defined inductively on programs statements, computing post-conditions:

$$[\![\mathsf{P}]\!]\varnothing \triangleq \varnothing \qquad [\![\mathsf{P}_1 \,;\, \mathsf{P}_2]\!]X \triangleq [\![\mathsf{P}_2]\!][\![\mathsf{P}_1]\!]X \qquad [\![\mathsf{skip}]\!]X \triangleq X$$

$$[\![\mathsf{x} := \mathsf{a}]\!]X \triangleq \{\mathsf{m}[\mathsf{x} \hookleftarrow (\!|\mathsf{a}|\!)\mathsf{m}] \mid \mathsf{m} \in X\}$$

$$[\![\mathsf{if}\ \mathsf{b}\ \{\mathsf{P}_1\}\ \mathsf{else}\ \{\mathsf{P}_2\}]\!]X \triangleq [\![\mathsf{P}_1]\!][\![\mathsf{b}]\!]X \cup [\![\mathsf{P}_2]\!][\![\neg\mathsf{b}]\!]X$$

$$[\![\mathsf{while}\ \mathsf{b}\ \{\mathsf{P}\}]\!]X \triangleq [\![\neg\mathsf{b}]\!](\mathit{lfp}^{\subseteq}_{\varnothing}\ \lambda T \,.\, X \cup [\![\mathsf{P}]\!][\![\mathsf{b}]\!]T)$$

Here $[\![\mathsf{b}]\!] \in \wp(\mathsf{Mem}) \to \wp(\mathsf{Mem})$ is the classic filtering function which selects the memories making b true, namely $[\![\mathsf{b}]\!]X \triangleq \{\mathsf{m} \in X \mid (\!|\mathsf{b}|\!)\mathsf{m} = \mathtt{tt}\}$. Note that $[\![\mathsf{P}]\!]$ computes exactly the strongest trace property of P, namely $[\![\mathsf{P}]\!]\mathsf{Mem} = {}^{\mathcal{R}}\mathcal{S}^{\mathsf{P}}$. Then we have to lift this semantics on sets of sets, obtaining a collecting hypersemantics $[\![\mathsf{P}]\!]^h \in \wp(\wp(\mathsf{Mem})) \to \wp(\wp(\mathsf{Mem}))$:

$$[\![\mathsf{P}]\!]^h\varnothing \triangleq \varnothing \qquad [\![\mathsf{P}_1 \,;\, \mathsf{P}_2]\!]^h\mathcal{X} \triangleq [\![\mathsf{P}_2]\!]^h[\![\mathsf{P}_1]\!]^h\mathcal{X} \qquad [\![\mathsf{skip}]\!]^h\mathcal{X} \triangleq \mathcal{X}$$

$$[\![\mathsf{x} := \mathsf{a}]\!]^h\mathcal{X} \triangleq \left\{[\![\mathsf{x} := \mathsf{a}]\!]X \;\middle|\; X \in \mathcal{X}\right\}$$

$$[\![\mathsf{if}\ \mathsf{b}\ \{\mathsf{P}_1\}\ \mathsf{else}\ \{\mathsf{P}_2\}]\!]^h\mathcal{X} \triangleq \left\{[\![\mathsf{P}_1]\!][\![\mathsf{b}]\!]X \cup [\![\mathsf{P}_2]\!][\![\neg\mathsf{b}]\!]X \;\middle|\; X \in \mathcal{X}\right\}$$

$$[\![\mathsf{while}\ \mathsf{b}\ \{\mathsf{P}\}]\!]^h\mathcal{X} \triangleq [\![\neg\mathsf{b}]\!]^h(\mathit{lfp}^{\subseteq}_{\varnothing}\ \mathsf{F}) \text{ where}$$

$$\mathsf{F} \triangleq \lambda\mathcal{T} \,.\, \mathcal{X} \cup \left\{[\![\mathsf{P}]\!][\![\mathsf{b}]\!]T \cup [\![\neg\mathsf{b}]\!]T \;\middle|\; T \in \mathcal{T}\right\}$$

This latter semantics is very similar to the *hypercollecting semantics* of [3], instantiated on sets of sets of memories instead of sets of sets of input/output traces. Again, $[\![\mathsf{b}]\!]^h \in \wp(\wp(\mathsf{Mem})) \to \wp(\wp(\mathsf{Mem}))$ is a filtering function, namely $[\![\mathsf{b}]\!]^h\mathcal{X} \triangleq \{\{\mathsf{m} \in X \mid (\!|\mathsf{b}|\!)\mathsf{m} = \mathtt{tt}\} \mid X \in \mathcal{X}\} \setminus \{\varnothing\}$. In this case $[\![\mathsf{P}]\!]^h$ is not exactly the strongest hyperproperty of P, but it is a correct approximation, i.e., ${}^{\mathcal{R}}\mathcal{HS}^{\mathsf{P}} \subseteq [\![\mathsf{P}]\!]^h\{\mathsf{Mem}\}$ [17]. Luckily, for subset-closed hyperproperties, as $\mathtt{equiv}^{\mathsf{L}}$, this is not a problem because $[\![\mathsf{P}]\!]^h$ is complete for such hyperproperties, namely $\mathsf{P} \models \mathtt{equiv}^{\mathsf{L}}$ iff $[\![\mathsf{P}]\!]^h\{\mathsf{Mem}\} \subseteq \mathtt{equiv}^{\mathsf{L}}$ [17]. Nevertheless, we are not interested in the strongest hyperproperty of P, since

[1] We abuse notation denoting with $\subseteq$ both set-inclusion and its component-wise extension to pairs, i.e., $\langle \mathcal{X}, \mathcal{Y}\rangle \subseteq \langle \mathcal{T}, \mathcal{Z}\rangle \triangleq (\mathcal{X} \subseteq \mathcal{T} \wedge \mathcal{Y} \subseteq \mathcal{X})$.

it is not necessary for proving Non-Interference. Instead, due to Prop. 3.4, it is sufficient to compute $\alpha_{nr}({}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=})_{\dashv}$ and this boils down to computing $[\![\mathsf{P}]\!]^h$ starting from $\mathcal{I}^{2,\mathsf{L}} \triangleq \{\{\mathsf{m}, \mathsf{m}'\} \mid \mathsf{m} =_{\mathsf{L}} \mathsf{m}'\}$. In fact, it is easy to see that $\alpha_{nr}({}^{\times}\mathcal{HS}^{\mathsf{P}\#2}_{=})_{\dashv}$ and $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}$ coincide with the set:

$$\left\{ \{\mathsf{m}_1', \mathsf{m}_2'\} \subseteq \mathsf{Mem} \;\middle|\; \exists \mathsf{m}_1, \mathsf{m}_2 \,.\; \begin{array}{l} \langle \mathsf{m}_1, \mathsf{P}\rangle \twoheadrightarrow^* \langle \mathsf{m}_1', \mathsf{skip}\rangle \wedge \\ \langle \mathsf{m}_2, \mathsf{P}\rangle \twoheadrightarrow^* \langle \mathsf{m}_2', \mathsf{skip}\rangle \wedge \\ \mathsf{m}_1 =_{\mathsf{L}} \mathsf{m}_2 \end{array} \right\}$$

This observation induces the following proposition.

Proposition 4.1. $\mathsf{P} \models \mathtt{NI}$ *iff* $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}} \subseteq \mathtt{equiv}^{\mathsf{L}}$.

4.2 Towards Abstraction

Unfortunately, $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}$ and $\mathtt{equiv}^{\mathsf{L}}$ are not computable in general, hence we need approximations. In order to compute a sound approximation of $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}$, we rely on abstract interpretation. The first step is to define the abstract domain used to verify Non-Interference.

4.2.1 The Abstract Domain for Non-Interference. The domain is an instance of the *hyperlevel (abstract) constants* of [17], and it checks whether a set of sets of values contains constant sets. In the following, all abstraction functions α are additive (i.e., they preserve the least upper bound of chains), hence their left adjoint α^- always exists[2]. From now on, we will often use the operator $(\ \mathsf{cond}\ ?\ \mathtt{doTrue} : \mathtt{doFalse}\ )$ in order to denote in-line conditional definitions, namely its semantics is: if the condition cond is true then substitute the operator with doTrue otherwise substitute it with doFalse. Mathematically:

$$(\ \mathsf{cond}\ ?\ \mathtt{doTrue} : \mathtt{doFalse}\ ) = \begin{cases} \mathtt{doTrue} & \text{if cond is true} \\ \mathtt{doFalse} & \text{otherwise} \end{cases}$$

Furthermore, we will sometimes use the pointwise extension of relations and operators. Given a relation $\preccurlyeq \subseteq Y \times Y$, an operator $\curlyvee \in Y \times Y \to Y$ and a set X, the pointwise extension $\dot{\preccurlyeq} \subseteq (X \to Y) \times (X \to Y)$, of $\preccurlyeq$, is defined as $f \mathrel{\dot{\preccurlyeq}} f' \triangleq (\forall x \in X \,.\, f(x) \preccurlyeq f'(x))$ and the pointwise extension $\dot{\curlyvee} \in (X \to Y) \times (X \to Y) \to (X \to Y)$, of $\curlyvee$, is defined as $f \mathrel{\dot{\curlyvee}} f' \triangleq \lambda x \in X \,.\, f(x) \curlyvee f'(x)$.

The domain is an abstraction of $\wp(\wp(\mathbb{Z}))$. In [17], the authors claim that a hyperdomain, i.e., a domain suitable for the verification of hyperproperties, can be decomposed in an *inner* abstraction, approximating traces, and in an *outer* abstraction, approximating properties of traces. Following this idea, we use as inner abstraction the one for the classic constant propagation domain: it represents precisely the singletons $\{n\}$, for every $n \in \mathbb{Z}$, and it abstracts to $\mathbb{Z}$ everything else. Instead, the outer abstraction checks whether the inner abstraction always returns constant values, not necessarily the same. For instance, $\{\{1\}, \{2\}\}$ contains only constant sets, instead $\{\{1\}, \{2, 3\}\}$ contains also a not-constant set (in this case the inner abstraction maps $\{2, 3\}$ to $\mathbb{Z}$).

We follow [17] defining $C_{\mathfrak{hc}} \triangleq \wp(\{\{n\} \mid n \in \mathbb{Z}\}) \cup \{\wp(\mathbb{Z})\}$, $\alpha_{\mathfrak{hc}} \triangleq \lambda\mathcal{X} \,.\, (\ \mathcal{X} \subseteq \{\{n\} \mid n \in \mathbb{Z}\}\ ?\ \mathcal{X} : \wp(\mathbb{Z})\ )$ and $\gamma_{\mathfrak{hc}} \triangleq id$. Then we have the Galois insertion:

$$\langle \wp(\wp(\mathbb{Z})), \subseteq\rangle \underset{\alpha_{\mathfrak{hc}}}{\overset{\gamma_{\mathfrak{hc}}}{\leftrightarrows}} \langle C_{\mathfrak{hc}}, \subseteq\rangle$$

This domain, as proved in [17], is sufficient for Non-Interference verification, but, as observed at the beginning of this section, it is not machine-representable. For this reason we need to perform a further approximation.

The domain $C_{\mathfrak{hc}}$ has an uncountable set of elements, so we define a simpler domain, which is machine-representable but still able to verify Non-Interference. Let $\bar{\mathbb{Z}}$ be a set isomorphic to $\mathbb{Z}$, aiming at representing sets containing only one singleton, i.e., of the form $\{\{n\}\}$, which is the information we want to observe precisely, and let $^{\bar{}} \in \mathbb{Z} \to \bar{\mathbb{Z}}$ a bijection. Furthermore, we denote by κ the abstract element representing the set of all singletons. Then we define $C^h \triangleq \{\bar{n} \mid n \in \mathbb{Z}\} \cup \{\bot, \top, \kappa\}$, with the partial order $\trianglelefteq \subseteq C^h \times C^h$ defined as $c_1 \trianglelefteq c_2 \triangleq (c_1 = \bot \vee c_1 = c_2 \vee (c_1 = \bar{n} \wedge c_2 = \kappa) \vee c_2 = \top)$. Now consider the abstraction $\alpha_{\hbar} \in C_{\mathfrak{hc}} \to C^h$:

$$\alpha_{\hbar}(\mathcal{X}) \triangleq \begin{cases} \bot & \text{if } \mathcal{X} = \varnothing \\ \bar{n} & \text{if } \mathcal{X} = \{\{n\}\} \\ \kappa & \text{if } \mathcal{X} \subseteq \{\{n\} \mid n \in \mathbb{Z}\} \wedge |\mathcal{X}| > 1 \\ \top & \text{otherwise} \end{cases}$$

Its corresponding concretization $\gamma_{\hbar} \in C^h \to C_{\mathfrak{hc}}$ is such that $\gamma_{\hbar}(\bot) = \varnothing$, $\gamma_{\hbar}(\bar{n}) = \{\{n\}\}$, $\gamma_{\hbar}(\kappa) = \{\{n\} \mid n \in \mathbb{Z}\}$ and $\gamma_{\hbar}(\top) = \wp(\mathbb{Z})$. Then we have the following Galois insertion:

$$\langle C_{\mathfrak{hc}}, \subseteq\rangle \underset{\alpha_{\hbar}}{\overset{\gamma_{\hbar}}{\leftrightarrows}} \langle C^h, \trianglelefteq\rangle$$

The domain $\langle C^h, \trianglelefteq, \triangledown, \triangle, \bot, \top\rangle$ is a compete lattice where $\triangledown, \triangle \in C^h \times C^h \to C^h$ are:

$$c_1 \triangledown c_2 \triangleq \begin{cases} \top & \text{if } c_1 = \top \vee c_2 = \top \\ \bot & \text{if } c_1 = \bot \wedge c_2 = \bot \\ \bar{n} & \text{if } \begin{array}{r} (c_1 = \bot \wedge c_2 = \bar{n}) \vee (c_1 = \bar{n} \wedge c_2 = \bot) \\ \vee\, (c_1 = \bar{n} \wedge c_2 = \bar{n}) \end{array} \\ \kappa & \text{otherwise} \end{cases}$$

$$c_1 \triangle c_2 \triangleq \begin{cases} \top & \text{if } c_1 = \top \wedge c_2 = \top \\ \bot & \text{if } c_1 = \bot \vee c_2 = \bot \vee (c_1 = \bar{n} \neq \bar{m} = c_2) \\ \bar{n} & \text{if } (c_1 = \bar{n} \wedge c_2 \in \{\kappa, \top\}) \vee (c_2 = \bar{n} \wedge c_1 \in \{\kappa, \top\}) \\ \kappa & \text{otherwise} \end{cases}$$

By composition, $\alpha_h \triangleq \alpha_{\hbar} \circ \alpha_{\mathfrak{hc}}$ and $\gamma_h \triangleq \gamma_{\mathfrak{hc}} \circ \gamma_{\hbar}$ form the insertion:

$$\langle \wp(\wp(\mathbb{Z})), \subseteq\rangle \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \langle C^h, \trianglelefteq\rangle$$

This domain approximates the set of sets of values a variable may have. Finally, in order to track information flows we need to work on memories instead of on values.

Consider a "double" non-relational abstraction for sets of sets of memories. Let $\dot{\mathsf{Mem}} \triangleq \mathsf{Var} \to \wp(\wp(\mathbb{Z}))$ and $\alpha_{nnr} \in \wp(\wp(\mathsf{Mem})) \to \dot{\mathsf{Mem}}$ be $\alpha_{nnr}(\mathcal{X}) \triangleq \lambda\mathsf{x} \,.\, \{\{\mathsf{m}(\mathsf{x}) \mid \mathsf{m} \in X\} \mid X \in \mathcal{X}\}$. We have the following Galois connection, with $\gamma_{nnr} \triangleq \alpha^-_{nnr}$:

$$\langle \wp(\wp(\mathsf{Mem})), \subseteq\rangle \underset{\alpha_{nnr}}{\overset{\gamma_{nnr}}{\leftrightarrows}} \langle \dot{\mathsf{Mem}}, \dot{\subseteq}\rangle$$

Now we can compose point-wise (the machine-representable version of) the hyperlevel constants abstraction with α_{nnr}, obtaining $\alpha_{\mathfrak{m}} \triangleq \lambda\mathcal{X} \,.\, \lambda\mathsf{x} \,.\, \alpha_h(\alpha_{nnr}(\mathcal{X})(\mathsf{x}))$. This latter forms, paired with $\gamma_{\mathfrak{m}} \triangleq \alpha^-_{\mathfrak{m}}$, the Galois connection:

$$\langle \wp(\wp(\mathsf{Mem})), \subseteq\rangle \underset{\alpha_{\mathfrak{m}}}{\overset{\gamma_{\mathfrak{m}}}{\leftrightarrows}} \langle \mathsf{Var} \to C^h, \dot{\trianglelefteq}\rangle$$

[2] This is a sufficient condition in order to form a Galois connection.

We denote with $\mathsf{Mem}^\sharp$ the set $\mathsf{Var} \to \mathcal{C}^h$ and we call its elements $\mathfrak{m}$ *abstract memories*. In order to simplify the notation, we let $\sqsubseteq \triangleq \dot{\trianglelefteq}$, $\sqcup \triangleq \dot{\triangledown}$, $\sqcap \triangleq \dot{\triangle}$, $\bot^\sharp \triangleq \lambda x\,.\,\bot$ and $\top^\sharp \triangleq \lambda x\,.\,\top$. We have that $\langle \mathsf{Mem}^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot^\sharp, \top^\sharp \rangle$ is a complete lattice. This latter is indeed our abstract hyperdomain of computation.

Example 4.2. Let us show in a simple example how this abstraction works. Continuing Ex. 3.3, we have: $\mathcal{I}^{2,\mathsf{L}} = \{\{a, b\}, \{c, d\}\}$ and $\alpha_\mathfrak{m}(\{\{a, b\}, \{c, d\}\}) = [x \mapsto \alpha_h(\{\{0\}, \{1\}\})\, y \mapsto \alpha_h(\{\{0, 1\}\})] = [x \mapsto \kappa\ y \mapsto \top]$. Indeed a, b both provide 0 to x and c, d both provide 1 to x, while a, b provide both 0 and 1 to y and c, d provide both 0 and 1 to y. Similarly, $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}} = \{\{b, d\}\}$ and $\alpha_\mathfrak{m}(\{\{b, d\}\}) = [x \mapsto \alpha_h(\{\{0, 1\}\})\ y \mapsto \alpha_h(\{\{1\}\})] = [x \mapsto \top\ y \mapsto \bar{1}]$. ∎

It is worth noting that, given $\mathcal{X} \subseteq \mathsf{equiv}^\mathsf{L}$, then every set in $\mathcal{X}$ contains L-equivalent memories. This implies $\alpha_\mathfrak{m}(\mathcal{X})(x) \neq \top$, for each L variable x. So, the Non-Interference check $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}} \subseteq \mathsf{equiv}^\mathsf{L}$ becomes equivalent to checking whether $\alpha_\mathfrak{m}([\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}})(x) \neq \top$, for each L variable x. Indeed the program of Ex. 4.2 is interferent, since computing $\alpha_\mathfrak{m}([\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}})$ results in L variable x having value $\top$.

Finally, we can show how the abstract domain $\mathsf{Mem}^\sharp$ can be used for Non-Interference verification, with the following theorem.

Theorem 4.3. *Let* $\mathfrak{m}^{\mathtt{NI}} \in \mathsf{Mem}^\sharp$ *be the abstract memory defined as* $\mathfrak{m}^{\mathtt{NI}}(x) \triangleq (\Gamma(x) = \mathsf{L}\,?\,\kappa : \top)$. *Then* $\mathsf{P} \models \mathtt{NI}$ *iff* $\alpha_\mathfrak{m}([\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq \mathfrak{m}^{\mathtt{NI}}$.

Proof. By definition of Galois connection: $\alpha_\mathfrak{m}([\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq \mathfrak{m}^{\mathtt{NI}}$ iff $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}} \subseteq \gamma_\mathfrak{m}(\mathfrak{m}^{\mathtt{NI}})$. It is clear that $\gamma_\mathfrak{m}(\mathfrak{m}^{\mathtt{NI}}) \subseteq \mathsf{equiv}^\mathsf{L}$, since the concretization of $\mathfrak{m}^{\mathtt{NI}}$ contains L-equivalent memories, by construction. Hence, due to Prop. 4.1, the theorem is proved. □

Hence, in order to verify Non-Interference it is sufficient an abstract semantics, computing on $\mathsf{Mem}^\sharp$, which approximates $[\![\mathsf{P}]\!]^h$. Indeed we can prove $\mathsf{P} \models \mathtt{NI}$ by computing an over-approximation of $[\![\mathsf{P}]\!]^h\mathcal{I}^{2,\mathsf{L}}$ in $\mathsf{Mem}^\sharp$.

4.2.2 The Abstract Semantics for Non-Interference. Finally, we have to show how to compute a program's collecting hypersemantics on the proposed abstract domain. The abstract semantics for programs relies on the abstract semantics for boolean and arithmetic expressions, given in Fig. 3,4. The abstract semantics for arithmetic expressions $(\!|a|\!)^\sharp \in \mathsf{Mem}^\sharp \to \mathcal{C}^h$ evaluates to an abstract value and it relies on the abstract (mathematical) operations given in Fig. 3. This semantics is such that abstract assignments are sound approximations of the concrete ones. We obtain this defining abstract operations $\oplus^\sharp$ such that they are sound w.r.t. the concrete ones $\oplus$, with $\oplus \in \{+, -, *\}$. This technically means that they satisfy the following constraint[3]:

$$\{\{X \ni n \oplus m \in Y\} \mid X \in \gamma_h(c_1), Y \in \gamma_h(c_2)\} \subseteq \gamma_h(c_1 \oplus^\sharp c_2)$$

The constraint basically requires that every possible result obtained applying the concrete operation is contained in the concretization of the application of the abstract operator.

The abstract semantics for booleans $(\!|b|\!)^\sharp \in \mathsf{Mem}^\sharp \to \mathsf{Mem}^\sharp$ is an abstract filtering function. To simplify, we assume that all negations $\neg$ have been removed using DeMorgan's laws and usual arithmetic laws: $\neg(b_1 \vee b_2) \equiv (\neg b_1) \wedge (\neg b_2)$, $\neg(a_1 < a_2) \equiv (a_2 \leq a_1)$,

[3]Recall that our abstract semantics is built after the "double" non-relational abstraction α_{nnr}, hence it abstracts $\mathsf{Mem} = \mathsf{Var} \to \wp(\wp(\mathbb{Z}))$.

Arithmetic expressions: $(\!|a|\!)^\sharp \in \mathsf{Mem}^\sharp \to \mathcal{C}^h$ with $\oplus \in \{+, -, *\}$

$$(\!|n|\!)^\sharp\mathfrak{m} \triangleq \bar{n} \qquad (\!|x|\!)^\sharp\mathfrak{m} \triangleq \mathfrak{m}(x) \qquad (\!|(a)|\!)^\sharp\mathfrak{m} \triangleq (\!|a|\!)^\sharp\mathfrak{m}$$

$$(\!|a_1 \oplus a_2|\!)^\sharp\mathfrak{m} \triangleq (\!|a_1|\!)^\sharp\mathfrak{m} \oplus^\sharp (\!|a_2|\!)^\sharp\mathfrak{m}$$

Abstract operations: $\oplus^\sharp \in \mathcal{C}^h \times \mathcal{C}^h \to \mathcal{C}^h$

$\oplus^\sharp$	$\bot$	$\bar{n}$	κ	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\bar{m}$	$\bot$	$\overline{m \oplus n}$	κ	$\top$
κ	$\bot$	κ	κ	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

Figure 3: Abstract semantics for arithmetic expressions.

Boolean expressions: $(\!|b|\!)^\sharp \in \mathsf{Mem}^\sharp \to \mathsf{Mem}^\sharp$ $\bowtie \in \{=, \neq, <, \leq\}$

$$(\!|\mathtt{tt}|\!)^\sharp\mathfrak{m} \triangleq \mathfrak{m} \qquad (\!|\mathtt{ff}|\!)^\sharp\mathfrak{m} \triangleq \bot^\sharp \qquad (\!|(b)|\!)^\sharp\mathfrak{m} \triangleq (\!|b|\!)^\sharp\mathfrak{m}$$

$$(\!|b_1 \vee b_2|\!)^\sharp\mathfrak{m} \triangleq \begin{array}{l}\text{let } \mathfrak{n} = (\!|b_1|\!)^\sharp\mathfrak{m} \sqcup (\!|b_2|\!)^\sharp\mathfrak{m} \text{ in} \\ \lambda x\,.\,(\,\mathfrak{m}(x) = \top \wedge x \in \mathsf{vars}(b_1) \cap \mathsf{vars}(b_2)\,?\,\top : \mathfrak{n}(x)\,)\end{array}$$

$$(\!|b_1 \wedge b_2|\!)^\sharp\mathfrak{m} \triangleq (\!|b_1|\!)^\sharp\mathfrak{m} \sqcap (\!|b_2|\!)^\sharp\mathfrak{m}$$

$$(\!|a_1 \bowtie a_2|\!)^\sharp\mathfrak{m} \triangleq \begin{array}{l}\text{let } \langle c_1, c_2\rangle = (\!|a_1|\!)^\sharp\mathfrak{m} \bowtie^\sharp (\!|a_2|\!)^\sharp\mathfrak{m} \text{ in} \\ \bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|a_1|\!)^\sharp\mathfrak{n} \trianglelefteq c_1\} \sqcap \bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|a_2|\!)^\sharp\mathfrak{n} \trianglelefteq c_2\}\end{array}$$

Abstract comparators $\bowtie^\sharp \in \mathcal{C}^h \times \mathcal{C}^h \to \mathcal{C}^h \times \mathcal{C}^h$:

$$c_1 \bowtie^\sharp c_2 \triangleq \begin{cases} \langle \bot, \bot\rangle & \text{if } c_1 = \bot \text{ or } c_2 = \bot \\ \langle \bar{n}, \bar{n}\rangle & \text{if } c_1 = c_2 = \bar{n}, \bowtie \in \{=, \leq\} \\ \langle \bar{n}, \bar{m}\rangle & \text{if } c_1 = \bar{n},\ c_2 = \bar{m},\ n < m,\ \bowtie \in \{<, \leq, \neq\} \\ \langle \bar{n}, \bar{m}\rangle & \text{if } c_1 = \bar{n},\ c_2 = \bar{m},\ n > m,\ \bowtie \in \{\neq\} \\ \langle \top, \top\rangle & \text{otherwise} \end{cases}$$

Figure 4: Abstract semantics for boolean expressions.

etc. This semantics must be sound w.r.t. the collecting hypersemantics for booleans, namely $[\![b]\!]^h\gamma_\mathfrak{m}(\mathfrak{m}) \subseteq \gamma_\mathfrak{m}((\!|b|\!)^\sharp)$. In order to obtain this, we have defined the abstract comparators $\bowtie^\sharp$ such that they are sound w.r.t. the concrete ones $\bowtie$, with $\bowtie \in \{=, \neq, <, \leq\}$. This technically means that they satisfy the following constraint:

$$\begin{array}{l}\text{let } \mathcal{X} = \{\{\langle n \in X, m \in Y\rangle \mid n \bowtie m\} \mid X \in \gamma_h(c_1), Y \in \gamma_h(c_2)\} \text{ in} \\ \quad \{\{n \mid \langle n, m\rangle \in X\} \mid X \in \mathcal{X}\} \subseteq \gamma_h((c_1 \bowtie^\sharp c_2)_\vdash) \\ \quad \{\{m \mid \langle n, m\rangle \in X\} \mid X \in \mathcal{X}\} \subseteq \gamma_h((c_1 \bowtie^\sharp c_2)_\dashv)\end{array}$$

The constraint basically requires that every possible pair of values making true the concrete comparator is contained in the concretization of the application of the abstract comparator. In Fig. 4, the term $\bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|a|\!)^\sharp\mathfrak{n} \trianglelefteq c\}$ can be computed with a *backward* abstract semantics for expressions (as we actually do in the implementation) but we omit here its definition for space reasons. The function vars(b) returns the set of variables occurring in b.

Example 4.4. Let us see how to compute $(\!|x < 2|\!)^\sharp\mathfrak{m}$ where $\mathfrak{m} = [x \mapsto \bar{1}\ y \mapsto \kappa]$. We have $(\!|x|\!)^\sharp\mathfrak{m} = \bar{1}$, $(\!|2|\!)^\sharp\mathfrak{m} = \bar{2}$ and $\bar{1} <^\sharp \bar{2} = \langle \bar{1}, \bar{2}\rangle$. Then $\bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|x|\!)^\sharp\mathfrak{n} \trianglelefteq \bar{1}\} = [x \mapsto \bar{1}\ y \mapsto \kappa]$ and $\bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|2|\!)^\sharp\mathfrak{n} \trianglelefteq \bar{2}\} = [x \mapsto \bar{1}\ y \mapsto \kappa]$. The intersection is, indeed, equal to $\mathfrak{m}$. Suppose now to compute the negation of $x < 2$, namely we want to compute $(\!|2 \leq x|\!)^\sharp\mathfrak{m}$. In this case we have $\bar{2} \leq^\sharp \bar{1} = \langle \top, \top\rangle$ and $\bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|2|\!)^\sharp\mathfrak{n} \trianglelefteq \top\} = \bigsqcup\{\mathfrak{n} \sqsubseteq \mathfrak{m} \mid (\!|x|\!)^\sharp\mathfrak{n} \trianglelefteq \top\} = [x \mapsto \bar{1}\ y \mapsto \kappa]$. Hence we obtain $[x \mapsto \bar{1}\ y \mapsto \kappa]$ as result. ∎

Finally, we need two auxiliary functions $\mathrm{vars}^{\top}_{\mathfrak{m}}(\mathsf{b})$ and $\mathrm{vars}^{:=}(\mathsf{P})$, returning the set of variables occurring in b having value $\top$ when evaluated in $\mathfrak{m}$ and the set of *modified variables* in P, respectively. The first is easy to compute: $\mathrm{vars}^{\top}_{\mathfrak{m}}(\mathsf{b}) \triangleq \{x \in \mathrm{vars}(\mathsf{b}) \mid \mathfrak{m}(x) = \top\}$ and vars(b) is just a syntactic check. The second involves semantic information, hence it is not trivial to compute. At the moment, we adopt a simple syntactic approach for approximating the set of variables which may be modified during P executions. Indeed, the function $\mathrm{vars}^{:=}(\mathsf{P})$ returns the set of variables occurring in P on the left-hand side of an assignment, which is easy implementable as a syntactic check. We plan to enhance our abstract semantics with a semantic check for modified variables, in order to increase precision, as a future work.

Now we have all the ingredients needed to define the abstract collecting hypersemantics $[\![\mathsf{P}]\!]^h_\sharp \in \mathsf{Mem}^\sharp \to \mathsf{Mem}^\sharp$, which is:

$$[\![\mathsf{P}]\!]^h_\sharp \bot^\sharp \triangleq \bot^\sharp \qquad [\![\mathsf{P}_1 \,;\, \mathsf{P}_2]\!]^h_\sharp \mathfrak{m} \triangleq [\![\mathsf{P}_2]\!]^h_\sharp [\![\mathsf{P}_1]\!]^h_\sharp \mathfrak{m}$$

$$[\![\mathsf{skip}]\!]^h_\sharp \mathfrak{m} \triangleq \mathfrak{m} \qquad [\![\mathsf{x} := \mathsf{a}]\!]^h_\sharp \mathfrak{m} \triangleq \mathfrak{m}[\mathsf{x} \hookleftarrow (\!|\mathsf{a}|\!)^\sharp \mathfrak{m}]$$

$[\![\mathsf{if\ b\ \{P_1\}\ else\ \{P_2\}}]\!]^h_\sharp \mathfrak{m} \triangleq \mathfrak{w}$ where

$$\text{let } \mathfrak{n} = [\![\mathsf{P}_1]\!]^h_\sharp (\!|\mathsf{b}|\!)^\sharp \mathfrak{m} \sqcup [\![\mathsf{P}_2]\!]^h_\sharp (\!|\neg\mathsf{b}|\!)^\sharp \mathfrak{m} \text{ in}$$

$$\mathfrak{w} = \lambda \mathsf{x}\,. \begin{cases} \mathfrak{n}(\mathsf{x}) & \text{if } \begin{array}{l} \mathsf{x} \notin \mathrm{vars}^{:=}(\mathsf{P}_1) \cup \mathrm{vars}^{:=}(\mathsf{P}_2) \vee \\ (\mathfrak{n}(\mathsf{x}) \trianglelefteq \kappa \wedge \mathfrak{n}(\mathsf{x}) \neq \kappa) \vee \mathrm{vars}^{\top}_{\mathfrak{m}}(\mathsf{b}) = \varnothing \end{array} \\ \top & \text{otherwise} \end{cases}$$

$$[\![\mathsf{while\ b\ \{P\}}]\!]^h_\sharp \mathfrak{m} \triangleq (\!|\neg\mathsf{b}|\!)^\sharp (\mathit{lfp}^{\sqsubseteq}_{\bot^\sharp}\, \lambda \mathfrak{n}.\mathfrak{m} \sqcup [\![\mathsf{if\ b\ \{P\}\ else\ \{skip\}}]\!]^h_\sharp \mathfrak{n})$$

The abstract semantics is quite standard for all statements, except for conditionals. We will explain here only this latter, which exploits the following idea. For every variable, we make the join between its value resulting after the execution of the true branch and its value resulting after the execution of the false branch. This is done in order to track the forbidden flows (implicit or explicit) generated inside the two branches. In fact, a L variable has value $\top$ after the join if in at least one of the branches it has value $\top$ (meaning that there is a forbidden flow). After this check we need to take in consideration the implicit flows generated by the conditional statement itself. Indeed, first we suppose that if there is at least one variable with value $\top$ *before* the boolean guard is evaluated, then all variables modified in the conditional branches have a forbidden flow (a variable has value $\top$ only if it is a H variable or if it has been "influence" by a H variable). This is done setting to $\top$ all modified variables. Note that if, for some reasons, a H variable is not $\top$ during this check, the flow is correctly not set. This procedure is sound but not so precise. In order to enhance precision, we exploit our abstract domain. In particular, we do not set to $\top$ the variables which have the same constant value $\bar{n}$ in both branches (this is the condition $(\mathfrak{n}(\mathsf{x}) \trianglelefteq \kappa \wedge \mathfrak{n}(\mathsf{x}) \neq \kappa)$) because this means that at the end of the conditional statement the variable has always a constant value. If there are no $\top$-valued variables into b (this is the condition $\mathrm{vars}^{\top}_{\mathfrak{m}}(\mathsf{b}) = \varnothing$), then no variables are set to $\top$: the resulting flows are those generated into the two branches of the conditional.

We proved that our abstract semantics is sound w.r.t. the collecting hypersemantics, and that it can be used for Non-Interference verification. This is stated in the following two theorems.

Theorem 4.5 (Soundness). *The abstract collecting hypersemantics is a sound approximation of the collecting hypersemantics:*

$$\forall \mathcal{X} \in \wp(\wp(\mathsf{Mem}))\,.\, \alpha_{\mathfrak{m}}([\![\mathsf{P}]\!]^h \mathcal{X}) \sqsubseteq [\![\mathsf{P}]\!]^h_\sharp \alpha_{\mathfrak{m}}(\mathcal{X})$$

Proof. (Sketch) We just have to prove that the abstract collecting hypersemantics approximates the best correct approximation of the collecting hypersemantics in $\mathsf{Mem}^\sharp$, namely $\alpha_{\mathfrak{m}} [\![\mathsf{P}]\!]^h \gamma_{\mathfrak{m}} \mathrel{\dot{=}} [\![\mathsf{P}]\!]^h_\sharp$. The proof relies on the soundness of the abstract semantics for arithmetic and boolean expressions and it is for structural induction. We omit here the full proof, we just show as example the case for programs sequences:

$$\begin{array}{ll} & \alpha_{\mathfrak{m}} [\![\mathsf{P}_1 \,;\, \mathsf{P}_2]\!]^h \gamma_{\mathfrak{m}}(\mathfrak{m}) \\ = & \quad \| \text{ definition of } [\![\cdot]\!]^h \\ & \alpha_{\mathfrak{m}} [\![\mathsf{P}_2]\!]^h [\![\mathsf{P}_1]\!]^h \gamma_{\mathfrak{m}}(\mathfrak{m}) \\ \sqsubseteq & \quad \| \text{ extensivity of } \gamma_{\mathfrak{m}} \alpha_{\mathfrak{m}} \\ & \alpha_{\mathfrak{m}} [\![\mathsf{P}_2]\!]^h \gamma_{\mathfrak{m}} \alpha_{\mathfrak{m}} [\![\mathsf{P}_1]\!]^h \gamma_{\mathfrak{m}}(\mathfrak{m}) \\ \sqsubseteq & \quad \| \text{ inductive hypothesis: } \alpha_{\mathfrak{m}} [\![\mathsf{P}_i]\!]^h \gamma_{\mathfrak{m}} \dot{\sqsubseteq} [\![\mathsf{P}_i]\!]^h_\sharp \text{ with } i \in \{1,2\} \\ & [\![\mathsf{P}_2]\!]^h_\sharp [\![\mathsf{P}_1]\!]^h_\sharp (\mathfrak{m}) \\ = & \quad \| \text{ definition of } [\![\cdot]\!]^h_\sharp \\ & [\![\mathsf{P}_1 \,;\, \mathsf{P}_2]\!]^h_\sharp (\mathfrak{m}) \end{array}$$

□

. With the abstract semantics just introduced we can define an effective verification method for Non-Interference.

Theorem 4.6 (Non-Interference Verification). *We have that a program satisfies Non-interference* $\mathsf{P} \models \mathsf{NI}$ *if* $[\![\mathsf{P}]\!]^h_\sharp \mathfrak{m}^{\mathsf{NI}} \sqsubseteq \mathfrak{m}^{\mathsf{NI}}$.

Proof. Note that $\alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq \mathfrak{m}^{\mathsf{NI}}$ since $\mathcal{I}^{2,\mathsf{L}}$ contains only sets of memories agreeing on L variables. This means that $\alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}})(\mathsf{x}) \trianglelefteq \kappa = \mathfrak{m}^{\mathsf{NI}}(\mathsf{x})$, for each L variable x. For H variables y, $\alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}})(\mathsf{y}) \trianglelefteq \top = \mathfrak{m}^{\mathsf{NI}}(\mathsf{y})$ trivially holds. Then the proof is given by the following implications:

$$\begin{array}{ll} & [\![\mathsf{P}]\!]^h_\sharp \mathfrak{m}^{\mathsf{NI}} \sqsubseteq \mathfrak{m}^{\mathsf{NI}} \\ \Downarrow & \quad \| \text{ monotonicity of } [\![\mathsf{P}]\!]^h_\sharp \text{ and } \alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq \mathfrak{m}^{\mathsf{NI}} \\ & [\![\mathsf{P}]\!]^h_\sharp \alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq [\![\mathsf{P}]\!]^h_\sharp \mathfrak{m}^{\mathsf{NI}} \sqsubseteq \mathfrak{m}^{\mathsf{NI}} \\ \Downarrow & \quad \| \text{ soundness of } [\![\mathsf{P}]\!]^h_\sharp \text{ (Thm. 4.5)} \\ & \alpha_{\mathfrak{m}}([\![\mathsf{P}]\!]^h \mathcal{I}^{2,\mathsf{L}}) \sqsubseteq [\![\mathsf{P}]\!]^h_\sharp \alpha_{\mathfrak{m}}(\mathcal{I}^{2,\mathsf{L}}) \sqsubseteq [\![\mathsf{P}]\!]^h_\sharp \mathfrak{m}^{\mathsf{NI}} \sqsubseteq \mathfrak{m}^{\mathsf{NI}} \\ \Downarrow & \quad \| \text{ Thm. 4.3} \\ & \mathsf{P} \models \mathsf{NI} \end{array}$$

□

This means that we can check Non-Interference simply by checking that each set of computations, starting from L-equivalent memories, provides only singletons as results.

5 THE PROTOTYPE ANALYZER

With the only aim of proving the feasibility of the proposed approach, and in particular of the abstract hypersemantics, we have written a prototype analyzer in Java SE 10 for Imp programs, which implements the abstract collecting semantics of Sec. 4. The tool,

```
Program P1:
a := 0 ;
if (b < x) { b := a * 3 }
else {b := a − ((2 * a) − a)}

Program P2:
a := x ;
if (b < a) { b := 2 }
else {b := 3}

Program P3:
a := x − x

Program P4:
a := 0 ;
while (x < y) {
    x := x + 1 ;
    while (a < x) {a := a + 2} ;
    a := 0
}
```

Figure 5: Example programs.

called nonInterfer, can be tested on-line, through a web interface, at the link http://bit.do/noninterfer.

5.1 Validation

Since our analyzer is built for a toy language, there are no benchmark test sets. So we have measured speed and precision of the tool building our own test set. We have written 25 non-interferent programs and 25 interferent programs, with different levels of complexity. As expected, the prototype does not output false negatives, i.e., all interferent programs are discovered. For what concerns precision, the analyzer marks 3 programs as interferent even if they actually satisfy Non-Interference. In Fig. 5 we have four example programs, where variables a, b are public and variables x, y are private. The initial abstract memory $\mathfrak{m}^{\mathtt{NI}}$ is $[\mathtt{a}\mapsto\kappa\ \mathtt{b}\mapsto\kappa\ \mathtt{x}\mapsto\top\ \mathtt{y}\mapsto\top]$. We have that $[\![\mathtt{P}_1]\!]^h_\natural \mathfrak{m}^{\mathtt{NI}} = [\mathtt{a}\mapsto\bar{0}\ \mathtt{b}\mapsto\bar{0}\ \mathtt{x}\mapsto\top\ \mathtt{y}\mapsto\top] \sqsubseteq \mathfrak{m}^{\mathtt{NI}}$, meaning that the analyzer correctly marks $\mathtt{P}_1$ as non-interferent. Analyzing program $\mathtt{P}_2$, the verifier is able to catch an implicit indirect flow, in fact $[\![\mathtt{P}_2]\!]^h_\natural \mathfrak{m}^{\mathtt{NI}} = [\mathtt{a}\mapsto\top\ \mathtt{b}\mapsto\top\ \mathtt{x}\mapsto\top\ \mathtt{y}\mapsto\top] \not\sqsubseteq \mathfrak{m}^{\mathtt{NI}}$ (i.e., $\mathtt{P}_2$ is correctly marked as interferent). Unfortunately, our analyzer signals a false alarm in program $\mathtt{P}_3$, indeed $[\![\mathtt{P}_3]\!]^h_\natural \mathfrak{m}^{\mathtt{NI}} = [\mathtt{a}\mapsto\top\ \mathtt{b}\mapsto\kappa\ \mathtt{x}\mapsto\top\ \mathtt{y}\mapsto\top] \not\sqsubseteq \mathfrak{m}^{\mathtt{NI}}$, even if the program is non-interferent. Finally, we have a precise result on the more complex program $\mathtt{P}_4$, indeed $[\![\mathtt{P}_4]\!]^h_\natural \mathfrak{m}^{\mathtt{NI}} = [\mathtt{a}\mapsto\bar{0}\ \mathtt{b}\mapsto\kappa\ \mathtt{x}\mapsto\top\ \mathtt{y}\mapsto\top] \sqsubseteq \mathfrak{m}^{\mathtt{NI}}$.

The finite height of the hyper abstract domain C^h guarantees the termination of the analysis. Furthermore, the structure of the domain allows us to compute loops fixpoints quickly, hence there is no need for a widening operator in order to speed-up the analysis. On our test set, which comprises quite small hand-made programs, the analyzer is very fast. The analysis time is around 120 milliseconds in average, running on a commodity hardware[4]. We also tested the prototype on bigger programs, generated automatically with the tool Grammarinator[15]. The analyzer is able to handle programs with hundreds of lines of code, basically with the same speed time. As expected, the analyzer shows some slowdowns when programs use lots of variables. Nevertheless, its running time is lower than 600 milliseconds even on programs with more than 500 variables.

Indeed the analyzer exhibits a good trade-off between verification speed and precision (in general). Unfortunately, our analyzer is not precise in some trivial situations, like in $\mathtt{P}_3$, hence in the next subsection we show how it is possible to get better results.

[4]Laptop with Arch Linux 64-bit (kernel 4.17.5-1), Intel Core i7-7700HQ CPU, 8GiB RAM and SSD storage.

5.2 Improving Precision

The analyzer has a good precision overall but it signals false alarms in some, sometimes very trivial, cases. We mentioned in the previous section that our current approach for the approximation of modified variables is a very simple syntactic check. With a more semantic analysis we can gain precision and do not arise false alarms for programs like $\mathtt{P}_3$ of Fig. 5. Apart from this detail, the sources of imprecision of our semantics are basically two: the approximation added making the hyperlevel constants domain machine-representable and the lack of relational information between variables. In this section we deal with these two issues.

5.2.1 Tuning the Hyperlevel Constants Domain. The original hyperlevel constants domain of [17] contains all the elements of the powerset of $\{\{n\} \mid n \in \mathbb{Z}\}$, meaning that every possible combination of constant sets is taken into account. This makes the domain very precise but not machine-representable, as already observed. In our implementation we have chosen to represent precisely only the singletons $\{\{n\}\}$, abstracted to $\bar{n}$, and the set of all singleton sets $\{\{n\} \mid n \in \mathbb{Z}\}$, abstracted to κ. In order to enhance precision we could extend our domain C^h with pairs of constant sets, namely we can represent sets of the form $\{\{n\},\{m\}\}$, with $n, m \in \mathbb{Z}$. But we can gain more precision taking into account triples, quadruples and so on. Hence we can infinitely tune the precision of the analyzer. Clearly the more elements we add to the domain and the more space is consumed by the analyzer and the more abstract operations are complex. So, the trade-off between precision and performance of analysis depends on the analyzer's context of application.

5.2.2 Add Relational Information. Our analysis is not-relational, meaning that we do not explicitly track relations between different variables. We can increase the precision pairing $\mathtt{Mem}^\natural$ with a relational abstraction of $\wp(\wp(\mathtt{Mem}))$. For instance we can define an abstract domain tracking equalities between variables. This latter, combined with a numerical domain such as the one for intervals will improve the precision w.r.t. implicit flows.

```
b := 1 ;
if (b = x) {
    a := 3 ;
    while (a! = 1) {
        a := a − 1 ;
        b := a
    }
} else {a := 1}
[a↦L b↦L x↦H]
```

Take, as example, the program here on the left. Our analyzer signals a false alarm, since the program is non-interferent but our analysis outputs an abstract memory assigning $\top$ to all variables. With an interval analysis we are able to find that variable a is equal to $[1, 1]$ at the end of the while and with a domain tracking equalities we can deduce the same for variable b. Hence, at the end of the program we can better our analysis obtaining the abstract memory $[\mathtt{a}\mapsto\bar{1}\ \mathtt{b}\mapsto\bar{1}\ \mathtt{x}\mapsto\top]$, allowing us to prove that the program is non-interferent.

6 RELATED WORK

The closest related works are [3] and [20], which both deal with hyperproperties by means of abstract interpretation. In the second, the authors define a hyperproperty called Input Data Usage, claiming that it generalizes a lot of notion of information flows, comprising Non-Interference. They propose an ad-hoc hypersemantics useful to verify that hyperproperty. Then they show how it is possible

to obtain, by abstraction of their semantics, some known verification methods for information flows. The work of [3] is focused on Non-Interference, indeed they propose two abstract semantics: one for (qualitative) Non-Interference and one for quantitative Non-Interference. Quantitative information flows go beyond the scope of the present work, hence we do not take in consideration that case. For the other case, we have that our abstract domain is not directly comparable with the "dependences" abstract domain of [3]. Nevertheless our abstract semantics is able to state correctly the non-interference of the program in Listing 5 of [3] without any tuning for precision. The dependences abstract semantics of [3] needs to add the Intervals domain in order to reach this level of precision. Finally, to the best of our knowledge, none of the previous works have an implementation, even for a toy language, supporting their theoretical results.

Another related work is [2], where authors propose a new methodology for proving the absence of timing channels. This work is based on the idea of "decomposition instead of self-composition". The idea is to partition the program semantics and to analyze each partition with standard methods. Their approach is similar to our method for simplifying the verification of Non-Interference. Nevertheless, for each partition they verify a classic trace property, instead we verify a hyperproperty. This lead us to better results w.r.t. precision.

Non-Interference is a k-hypersafety (with $k = 2$) hence it can be verified with a classic mechanism for safety trace properties on the k times self-composed system. The self-composition can be sequential, parallel or in an interleaving manner and a lot of works applied this methodology [5, 18, 19]. Unfortunately, this approach seems to be computationally to expensive to be used in practice [2]. Besides the reduction to safety, in [1] the authors introduce a runtime refutation method for k-hypersafety, based on a three-valued logic. Similarly, [6, 13] define hyperlogics (HyperLTL and HyperCTL/CTL*), i.e., extensions of temporal logic able to quantify over multiple traces. Some algorithms for model-checking in these extended temporal logics exist, but only for particular decidable fragments, since the model-checking problem for these logics is, in general, undecidable.

Classic methods for Non-Interference verification, which do not take in consideration its hyperproperty nature, comprise the type systems à la Volpano [21]. These latter perform just syntactic checks, with our approach we have more precision since we can exploit semantic information. The new logic-based approaches showed up recently seem very promising, like the epistemic temporal logic of [4] and SecLTL [12]. They extend classic temporal logics with modalities useful for the verification of Non-Interference. It is not so easy to compare our work with these latter, we let as a future work to deepen the link between these logics and our abstract semantics.

7 CONCLUSION AND FUTURE DIRECTIONS

In this work, we show step by step how to build a static analyzer for Non-Interference, based on abstract interpretation. The tool is sound, meaning that it will not signals false negative, i.e., if the analyzer returns that a program is non-interferent then it is guaranteed that it satisfies Non-Interference. We have soundness by design, exploiting the framework of abstract interpretation. We follow the theoretical work of [17] and we made their abstract domain computer-representable, hence we show how to make their analysis feasible. Furthermore, we simplified the process of Non-Interference verification, moving from a semantics computing on input/output traces to a simpler semantics computing on memories.

We implemented our analyzer, called nonInterfer in order to validate our abstract semantics. The testing on our prototype has lead to very promising results, in particular w.r.t. analysis speed. Non-Interference verification is undecidable, hence we have obviously false negative, namely sometimes our analyzer marks as interferent a program which actually satisfies Non-interference. Despite its simplicity and speed, our analyzer is quite precise, at least as precise as classic type systems for Non-Interference.

As a future work, we will increase the precision of nonInterfer, adding the possibility to track relational information between different variables. Finally, since the results on Imp are promising, we planned the porting of nonInterfer to a real-world programming language, in order to see its performance on more challenging tests.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] S. Agrawal and B. Bonakdarpour. 2016. Runtime Verification of k-Safety Hyperproperties in HyperLTL. In *Proc. of CSF*. 239–252.
[2] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proc. of PLDI*. 362–375.
[3] M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. In *Proc. of POPL*. 874–887.
[4] M. Balliu, M. Dam, and G. Le Guernic. 2011. Epistemic Temporal Logic for Information Flow Security. In *Proc. of PLAS*. 1–12.
[5] G. Barthe, P. R. D'Argenio, and T. Rezk. 2004. Secure information flow by self-composition. In *Proc. of CSF*. 100–114.
[6] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. 2014. Temporal Logics for Hyperproperties. In *Proc. of POST*.
[7] M. R. Clarkson and F. B. Schneider. 2010. Hyperproperties. *J. of Comp. Security* 18 (2010), 1157–1210.
[8] E. Cohen. 1977. Information Transmission in Computational Systems. *Oper. Syst. Rev.* 11 (1977), 133–139.
[9] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*. 238–252.
[10] P. Cousot and R. Cousot. 2012. An Abstract Interpretation Framework for Termination. In *Proc. of POPL*. 245–258.
[11] P. Cousot, R. Giacobazzi, and F. Ranzato. 2018. Program Analysis Is Harder Than Verification: A Computability Perspective. In *Proc. of CAV*. 75–95".
[12] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. 2012. Model Checking Information Flow in Reactive Systems. In *Proc. of VMCAI*. 169–185.
[13] B. Finkbeiner, M. N. Rabe, and C. Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL. In *Proc. of CAV*. 30–48.
[14] D. Hedin and A. Sabelfeld. 2012. A Perspective on Information-Flow Control. *Software Safety and Security* 33 (2012), 319–347.
[15] R. Hodovan and A. Kiss. 2018. Grammarinator - ANTLRv4 grammar-based test generator. https://github.com/renatahodovan/grammarinator
[16] I. Mastroeni and M. Pasqua. 2017. Hyperhierarchy of Semantics - A Formal Framework for Hyperproperties Verification. In *Proc. of SAS*. 232–252.
[17] I. Mastroeni and M. Pasqua. 2018. Verifying Bounded Subset-Closed Hyperproperties. In *Proc. of SAS*. 263–283.
[18] M. Sousa and I. Dillig. 2016. Cartesian Hoare Logic for Verifying K-safety Properties. In *Proc. of PLDI*. 57–69.
[19] T. Terauchi and A. Aiken. 2005. Secure Information Flow As a Safety Problem. In *Proc. of SAS*. 352–367.
[20] C. Urban and P. Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Proc. of ESOP*. 683–710.
[21] D. Volpano, C. Irvine, and G. Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4 (1996), 167–187.