Metadata of the chapter that will be visualized in SpringerLink

Book Title	Transactions on Computational Science XXXI	
Series Title		
Chapter Title	Combining Symbolic and Numerical Domains for Information Leakage Analysis	
Copyright Year	2018	
Copyright HolderName	Springer-Verlag GmbH Germany	
Corresponding Author	Family Name	**Cortesi**
	Particle	
	Given Name	**Agostino**
	Prefix	
	Suffix	
	Division	
	Organization	Ca’ Foscari University
	Address	Venice, Italy
	Email	cortesi@unive.it
	ORCID	http://orcid.org/0000-0002-0946-5440
Author	Family Name	**Ferrara**
	Particle	
	Given Name	**Pietro**
	Prefix	
	Suffix	
	Division	
	Organization	Julia srl
	Address	Verona, Italy
	Email	
Author	Family Name	**Halder**
	Particle	
	Given Name	**Raju**
	Prefix	
	Suffix	
	Division	
	Organization	Indian Institute of Technology Patna
	Address	Patna, India
	Email	
Author	Family Name	**Zanioli**
	Particle	
	Given Name	**Matteo**
	Prefix	
	Suffix	
	Division	
	Organization	Alpenite srl

Address Venice, Italy
Email

Abstract

We introduce an abstract domain for information-flow analysis of software. The proposal combines variable dependency analysis with numerical abstractions, yielding to accuracy and efficiency improvements. We apply the full power of the proposal to the case of database query languages as well. Finally, we present an implementation of the analysis, called Sails, as an instance of a generic static analyzer. Keeping the modular construction of the analysis, the tool allows one to tune the granularity of heap analysis and to choose the numerical domain involved in the reduced product. This way the user can tune the information leakage analysis at different levels of precision and efficiency.

Combining Symbolic and Numerical Domains for Information Leakage Analysis

Agostino Cortesi[1(✉)], Pietro Ferrara[2], Raju Halder[3], and Matteo Zanioli[4]

[1] Ca' Foscari University, Venice, Italy
cortesi@unive.it
[2] Julia srl, Verona, Italy
[3] Indian Institute of Technology Patna, Patna, India
[4] Alpenite srl, Venice, Italy

Abstract. We introduce an abstract domain for information-flow analysis of software. The proposal combines variable dependency analysis with numerical abstractions, yielding to accuracy and efficiency improvements. We apply the full power of the proposal to the case of database query languages as well. Finally, we present an implementation of the analysis, called Sails, as an instance of a generic static analyzer. Keeping the modular construction of the analysis, the tool allows one to tune the granularity of heap analysis and to choose the numerical domain involved in the reduced product. This way the user can tune the information leakage analysis at different levels of precision and efficiency. AQ1

1 Introduction

Protecting the confidentiality is a relevant problem when sensitive information flows through computing systems or transmits over public networks. Standard protection mechanisms, such as encryption, access control, etc. can suitably be applied at source level, but they are unable to protect the confidentiality once the information is released from the source and is allowed to flow through the computing systems.

The starting point of secure information flow analysis in software applications is the classification of program variables into different security levels. In the simplest case, two levels are commonly used: public (or low, L) and secret (or high, H). The main purpose is to prevent the leakage of sensitive information when flowing (implicitly or explicitly) from a high variable h to a lower one l. An explicit flow from h to l occurs when the content of h directly affects (e.g., through an assignment operator) l. On the other hand, an implicit flow from h to l occurs when the content of l gets affected indirectly (e.g., through a boolean condition in an `if` statement) by h, as stated in [17].

There is a widespread literature on methods and techniques for checking secure information flows in software. Generally, works on information flow fall into two categories: (i) dynamic, instrumentation based approaches ($e.g.$, tainting), and (ii) static, language-based approaches ($e.g.$, type systems). The dynamic approaches

M. L. Gavrilova et al. (Eds.): Trans. on Comput. Sci. XXXI, LNCS 10730, pp. 1–38, 2018.
https://doi.org/10.1007/978-3-662-56499-8_6

introduce significant run-time overhead [10,33]. The static approaches typically require some changes to the language and the run-time environment as well as non-trivial type annotations [38], making their adoption too expensive in practice.

Nevertheless, despite of these deep and extensive works, their practical applications have been relatively poor. Usually these approaches work on an ad-hoc programming language [4], and they do not support mainstream languages. This means that one should completely rewrite a program in order to apply them to some existing code.

Recently a new generic static analyzer (Sample[1]) based on the Abstract Interpretation theory has been developed and applied to many different contexts and analysis. Roughly, this analyzer splits and combines the abstraction of the heap and the approximation of other semantic information, *e.g.* string [12], type [19] abstractions.

In this paper[2], we introduce a language-based information-flow analysis of imperative and database query languages based on the Abstract Interpretation framework, by combining symbolic and numerical domains; we present the tool Sails (Static Analysis of Information Leakage with Sample); finally, we show experimental results applying Sails on security benchmark programs.

In particular,

1. we represent variables' dependences in the form of propositional formula $\psi = x \rightarrow y$, where x, y are variables and value of y possibly depend on the value of x; in order to detect possible information leakage, we check the satisfiability of ψ when assigning each variable the truth value corresponding to its sensitivity level;
2. we define abstract semantics of (i) imperative and (ii) database query languages in the domain of propositional formulae, by considering an over-approximation of variables' dependences at each program point;
3. we enhance the accuracy of the technique by analysing programs over numerical abstract domains, using reduced product of the symbolic propositional formulae domain and numerical abstract domains;
4. finally, we show encouraging experimental results on a set of security benchmarks using the tool Sails which is implemented based on our proposal.

The overall analysis combines a symbolic variable dependency analysis, based on the propositional formulae domain [11], and a variable value dependency analysis using numerical abstractions (e.g., intervals or polyhedra). Unlike other works, our proposal provides an information flow analysis without any major constraint on the target language, since it tracks information flows between variables and heap locations over programs written in mainstream object-oriented languages like Java and Scala.

The rest of the paper is organized as follows. Section 2 introduces the dependency analysis through the propositional formulae domain. Section 3 combines the dependency analysis with numerical domains through a reduced product.

[1] http://www.pm.inf.ethz.ch/research/semper/Sample.

[2] The paper is a revised and extended version of [25,47,48].

An extension to the case of database query languages is discussed in Sect. 4. Section 5 presents the main issues we solved in order to plug this information flow analysis into Sample while developing Sails. Section 6 presents the experimental results when applying Sails to a complex case study and to the SecuriBench-micro suite. Finally, Sect. 7 presents the related work and Sect. 8 concludes.

2 Dependency Analysis

This section formalizes the dependency analysis and proves its soundness following the abstract interpretation framework.

2.1 The Language

For the sake of simplicity, we consider a simple imperative language where programs consist of labeled commands (similar to [26]). The syntax is defined in Table 1[3].

Table 1. Syntax of the language

Expressions			
exp	$\in$	E	
exp	::=	n	where $n \in \mathbb{N}$
	|	v	
	|	$\mathsf{exp}_1 \oplus \mathsf{exp}_2$	where $\oplus = \{+, -, *, /\}$
Conditions			
b	$\in$	B	
b	::=	true	
	|	false	
	|	$\mathsf{b}_1 \otimes \mathsf{b}_2$	where $\otimes = \{\vee, \wedge\}$
	|	$\neg\mathsf{b}$	
	|	$\mathsf{exp}_1 \oslash \mathsf{exp}_2$	where $\oslash = \{\leq, >, =\}$
Labeled commands			
ℓ	$\in$	L	set of labels
c	$\in$	C	
c	::=	$^{\ell}\mathsf{skip}$	
	|	$^{\ell}\mathsf{v} := \mathsf{exp}$	
	|	if $^{\ell}\mathsf{b}$ then c_1 else c_2 $^{\ell'}$endif	
	|	$\mathsf{c}_1; \mathsf{c}_2$	
	|	while $^{\ell}\mathsf{b}$ do c $^{\ell'}$done	
$\mathcal{P}$	::=	c^{ℓ}	program that ends with label ℓ

[3] In the rest of the paper, we will omit the initial and final labels of statements when not required.

Table 2. Initial label function

$$
\begin{aligned}
in[\![{}^{\ell}\mathsf{skip}]\!] &\stackrel{\text{def}}{=} \ell\\
in[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!] &\stackrel{\text{def}}{=} \ell\\
in[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!] &\stackrel{\text{def}}{=} \ell\\
in[\![\mathsf{c}_1;\mathsf{c}_2]\!] &\stackrel{\text{def}}{=} in[\![\mathsf{c}_1]\!]\\
in[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!] &\stackrel{\text{def}}{=} \ell
\end{aligned}
$$

Table 3. Final label function

$\mathcal{P} ::= \mathsf{c}^{\ell}$	$fin[\![\mathcal{P}]\!] \equiv \ell$
	$fin[\![\mathsf{c}]\!] \equiv fin[\![P]\!]$
$\mathsf{c} ::= {}^{\ell}\mathsf{skip}$	$fin[\![{}^{\ell}\mathsf{skip}]\!] \equiv fin[\![\mathsf{c}]\!]$
$\mid\ {}^{\ell}\mathsf{v} := \mathsf{exp}$	$fin[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!] \equiv fin[\![\mathsf{c}]\!]$
$\mid\ \mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}$	$fin[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!] \equiv fin[\![\mathsf{c}]\!]$
	$fin[\![\mathsf{c}_1]\!] \equiv \ell'$
	$fin[\![\mathsf{c}_2]\!] \equiv \ell'$
$\mid\ \mathsf{c}_1;\mathsf{c}_2$	$fin[\![\mathsf{c}_1;\mathsf{c}_2]\!] \equiv fin[\![\mathsf{c}]\!]$
	$fin[\![\mathsf{c}_1]\!] \equiv in[\![\mathsf{c}_2]\!]$
	$fin[\![\mathsf{c}_2]\!] \equiv fin[\![\mathsf{c}]\!]$
$\mid\ \mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}$	$fin[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ \mathsf{done}]\!] \equiv fin[\![\mathsf{c}]\!]$
	$fin[\![\mathsf{c}]\!] \equiv \ell$

Table 4. Action function

$$
\begin{aligned}
a[\![{}^{\ell}\mathsf{skip}]\!] &\stackrel{\text{def}}{=} \{{}^{\ell}\mathit{skip}\}\\
a[\![{}^{\ell}\mathsf{v} := \mathsf{exp}]\!] &\stackrel{\text{def}}{=} \{{}^{\ell}\mathsf{v} := \mathsf{exp}\}\\
a[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}]\!] &\stackrel{\text{def}}{=} \{{}^{\ell}\mathsf{b},\ {}^{\ell}\mathsf{not\ b}, {}^{\ell'}\ \mathsf{endif}\} \cup a[\![\mathsf{c}_1]\!] \cup a[\![\mathsf{c}_2]\!]\\
a[\![\mathsf{c}_1;\mathsf{c}_2]\!] &\stackrel{\text{def}}{=} a[\![\mathsf{c}_1]\!] \cup a[\![\mathsf{c}_2]\!]\\
a[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}]\!] &\stackrel{\text{def}}{=} \{{}^{\ell}\mathsf{b}, {}^{\ell}\ \mathsf{not\ b}, {}^{\ell'}\ \mathsf{done}\} \cup a[\![\mathsf{c}]\!]
\end{aligned}
$$

Let $in : \mathsf{C} \to \mathsf{L}$ and $fin : \mathsf{C} \to \mathsf{L}$ be two functions. By $in[\![\mathsf{c}]\!]$ and $fin[\![\mathsf{c}]\!]$ we denote the *initial* and *final label* of command $\mathsf{c} \in \mathsf{C}$ respectively. These two functions are formally defined in Tables 2 and 3.

Each command corresponds to one or more *actions*. The set of actions, denoted by A, consists of $\{{}^{\ell}\mathsf{skip},\ {}^{\ell}\mathsf{v} := \mathsf{exp},\ {}^{\ell}\mathsf{b},\ {}^{\ell}\neg\mathsf{b},\ {}^{\ell}\mathsf{endif},\ {}^{\ell}\mathsf{done}\}$. Let $a : \mathsf{C} \to \wp(\mathsf{A})$ be a function that, given a command, returns the set of actions involved in it. The function a for various commands is defined in Table 4.

Without loss of generality, we assume that the variables appearing in a program are implicitly declared. We denote by $\mathsf{V}(\mathcal{P})$ the set of variables in program $\mathcal{P}$ and, similarly, by $\mathsf{V}(\mathsf{exp})$ and $\mathsf{V}(\mathsf{b})$ the variables contained in expression exp and condition b respectively. The definition of V is reported in Table 5.

Table 5. Variables functions

$$
\begin{aligned}
\mathsf{V}(n) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{V}(\mathsf{v}) &\stackrel{\text{def}}{=} \{\mathsf{v}\} \\
\mathsf{V}(\mathsf{exp}_1 \oplus \mathsf{exp}_2) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{exp}_1) \cup \mathsf{V}(\mathsf{exp}_2) \\
\hline
\mathsf{V}(\mathsf{true}) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{V}(\mathsf{false}) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{V}(\mathsf{b}_1 \otimes \mathsf{b}_2) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{b}_1) \cup \mathsf{V}(\mathsf{b}_2) \\
\mathsf{V}(\mathsf{exp}_1 \oslash \mathsf{exp}_2) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{exp}_1) \cup \mathsf{V}(\mathsf{exp}_2) \\
\hline
\mathsf{V}(\mathsf{skip}) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{V}(\mathsf{v} := \mathsf{exp}) &\stackrel{\text{def}}{=} \{\mathsf{v}\} \cup \mathsf{V}(\mathsf{exp}) \\
\mathsf{V}(\mathsf{if\ b\ then\ c_1\ else\ c_2\ endif}) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{b}) \cup \mathsf{V}(\mathsf{c}_1) \cup \mathsf{V}(\mathsf{c}_2) \\
\mathsf{V}(\mathsf{c}_1 ; \mathsf{c}_2) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{c}_1) \cup \mathsf{V}(\mathsf{c}_2) \\
\mathsf{V}(\mathsf{while\ b\ do\ c\ done}) &\stackrel{\text{def}}{=} \mathsf{V}(\mathsf{b}) \cup \mathsf{V}(\mathsf{c})
\end{aligned}
$$

Table 6. Evaluation of expressions

$$
\begin{aligned}
&\mathrm{E} \in \mathsf{E} \to (\mathcal{E} \to \mathbb{N}) \\
\mathrm{E}[\![n]\!]\rho &\stackrel{\text{def}}{=} n \\
\mathrm{E}[\![\mathsf{v}]\!]\rho &\stackrel{\text{def}}{=} \rho(\mathsf{v}) \\
\mathrm{E}[\![\mathsf{exp}_1 \oplus \mathsf{exp}_2]\!]\rho &\equiv v_1 \oplus v_2 \text{ (such that } v_1 = \mathrm{E}[\![\mathsf{exp}_1]\!]\rho \wedge v_2 = \mathrm{E}[\![\mathsf{exp}_2]\!]\rho)
\end{aligned}
$$

Table 7. Evaluation of boolean conditions

$$
\begin{aligned}
&\mathrm{B} \in \mathsf{B} \to (\mathcal{E} \to \{\mathsf{true}, \mathsf{false}\}) \\
\mathrm{B}[\![\mathsf{true}]\!]\rho &\stackrel{\text{def}}{=} \mathsf{true} \\
\mathrm{B}[\![\mathsf{false}]\!]\rho &\stackrel{\text{def}}{=} \mathsf{false} \\
\mathrm{B}[\![\mathsf{b}_1 \otimes \mathsf{b}_2]\!]\rho &\stackrel{\text{def}}{=} b_1 \otimes b_2 \text{ (such that } b_1 = \mathrm{B}[\![\mathsf{b}_1]\!]\rho \wedge b_2 = \mathrm{B}[\![\mathsf{b}_2]\!]\rho) \\
\mathrm{B}[\![\mathsf{exp}_1 \oslash \mathsf{exp}_2]\!]\rho &\stackrel{\text{def}}{=} \mathsf{true} \text{ if } \exists v_1 = \mathrm{E}[\![\mathsf{exp}_1]\!]\rho : v_2 = \mathrm{E}[\![\mathsf{exp}_2]\!]\rho : v_1 \oslash v_2 \\
&\quad\ \mathsf{false} \text{ if } \exists v_1 = \mathrm{E}[\![\mathsf{exp}_1]\!]\rho : v_2 = \mathrm{E}[\![\mathsf{exp}_2]\!]\rho : \mathsf{not}(v_1 \oslash v_2)
\end{aligned}
$$

2.2 The Concrete Domain

An *environment* $\rho \in \mathcal{E}$ is a function $\rho : \mathsf{V} \to \mathbb{N}$ which assigns a value to each variable. A *state* $\sigma \in \Sigma = (\mathsf{L} \times \mathcal{E})$ is a pair $\langle \ell, \rho \rangle$ where the program label ℓ is the label of the action to be executed and the environment ρ defines the values of program variables at ℓ.

We denote by $\mathrm{E}[\![\mathsf{exp}]\!]\sigma$ and $\mathrm{B}[\![\mathsf{b}]\!]\sigma$ the evaluation of expression $\mathsf{exp} \in \mathsf{E}$ and condition $\mathsf{b} \in \mathsf{B}$ respectively on the state σ. The details can be found in Tables 6 and 7 respectively.

Given a program $\mathcal{P}$, the set of possible initial and final states are defined as $\mathsf{I}[\![\mathcal{P}]\!] \equiv \left\{ \langle in[\![\mathcal{P}]\!], \rho \rangle \mid \rho \in \mathcal{E} \right\}$ and $\mathsf{F}[\![\mathcal{P}]\!] \equiv \left\{ \langle fin[\![\mathcal{P}]\!], \rho \rangle \mid \rho \in \mathcal{E} \right\}$.

Table 8. The transition function

$$
\begin{array}{l}
\mathsf{T}[\![{}^{\ell}\mathsf{skip}^{\ell_f}]\!] = \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{skip}^{\ell_f}} \langle \mathit{fin}[\![{}^{\ell}\mathsf{skip}^{\ell_f}]\!], \rho\rangle \mid \rho \in \mathcal{E}\}\\
\mathsf{T}[\![{}^{\ell}\mathsf{v} := \mathsf{exp}^{\ell_f}]\!] = \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{v:=exp}^{\ell_f}} \langle \mathit{fin}[\![{}^{\ell}\mathsf{v} := \mathsf{exp}^{\ell_f}]\!], \rho[\mathsf{v} \leftarrow v]\rangle \mid \rho \in \mathcal{E} \wedge v = \mathsf{E}[\![\mathsf{exp}]\!]\rho\}\\
\mathsf{T}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}^{\ell_f}]\!] = \mathsf{T}[\![\mathsf{c}_1]\!] \cup \mathsf{T}[\![\mathsf{c}_2]\!] \cup\\
\qquad \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle \mathit{in}[\![\mathsf{c}_1]\!], \rho\rangle \mid \rho \in \mathcal{E} \wedge\ \mathsf{true} = \mathsf{B}[\![\mathsf{b}]\!]\rho\} \cup\\
\qquad \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle \mathit{in}[\![\mathsf{c}_2]\!], \rho\rangle \mid \rho \in \mathcal{E} \wedge\ \mathsf{false} = \mathsf{B}[\![\mathsf{b}]\!]\rho\} \cup\\
\qquad \{\langle \ell', \rho\rangle \xrightarrow{{}^{\ell'}\mathsf{endif}^{\ell_f}} \langle \mathit{fin}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_1\ \mathsf{else}\ \mathsf{c}_2\ {}^{\ell'}\mathsf{endif}^{\ell_f}]\!], \rho\rangle \mid \rho \in \mathcal{E}\}\\
\mathsf{T}[\![\mathsf{c}_1; \mathsf{c}_2]\!] = \mathsf{T}[\![\mathsf{c}_1]\!] \cup \mathsf{T}[\![\mathsf{c}_2]\!]\\
\mathsf{T}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}^{\ell_f}]\!] = \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{not\ b}} \langle \ell', \rho\rangle \mid \rho \in \mathcal{E} \wedge \mathsf{false} = \mathsf{B}[\![\mathsf{b}]\!]\rho\} \cup\\
\qquad \{\langle \ell, \rho\rangle \xrightarrow{{}^{\ell}\mathsf{b}} \langle \mathit{in}[\![\mathsf{c}]\!], \rho\rangle \mid \rho \in \mathcal{E} \wedge \mathsf{true} = \mathsf{B}[\![\mathsf{b}]\!]\rho\} \cup \mathsf{T}[\![\mathsf{c}]\!] \cup\\
\qquad \{\langle \ell', \rho\rangle \xrightarrow{{}^{\ell'}\mathsf{done}} \langle \mathit{fin}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}^{\ell_f}]\!], \rho\rangle \mid \rho \in \mathcal{E}\}
\end{array}
$$

The *labeled transition semantics* $\mathsf{T}[\![\mathsf{c}]\!]$ of a command $\mathsf{c} \in \mathcal{P}$ is a set of transitions $\langle \sigma_1, \mathsf{a}, \sigma_2\rangle$ between a state σ_1 and its next states σ_2 by an action $\mathsf{a} \in a(\mathsf{c})$. The triple $\langle \sigma_1, \mathsf{a}, \sigma_2\rangle$ is also denoted by $\sigma_1 \xrightarrow{\mathsf{a}} \sigma_2$. The transition function $\mathsf{T} : \mathsf{C} \rightarrow \wp(\Sigma \times \mathsf{A} \times \Sigma)$ in Table 8 tracks all reachable states.

A *labeled transition system* is a tuple $\langle \Sigma, \mathsf{I}, \mathsf{F}, \mathsf{A}, \mathsf{T}\rangle$, where Σ is the set of states, $\mathsf{I} \subseteq \Sigma$ is a nonempty set of initial states, $\mathsf{F} \subseteq \Sigma$ is a set of final states, A is a nonempty set of actions, and $\mathsf{T} \in \wp(\Sigma \times \mathsf{A} \times \Sigma)$ is the labeled transition relation.

We define the *partial trace semantics* of a transition system, similarly to [26], as the set of all possible traces of elements in Σ (denoted by $\Sigma^\star$), recording the observation of executions starting from initial states and possibly reaching final states in finite time.

$$\Sigma^\star \in \wp(\Sigma \times \mathsf{A} \times \Sigma)$$
$$\Sigma^\star = \{\sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \mid n \geq 1 \wedge \sigma_0 \in \mathsf{I} \wedge \forall i \in [0, n-1] : \sigma_i \xrightarrow{\mathsf{a}_i} \sigma_{i+1} \in T^{\ell}\}$$

Let $\pi_0, \pi_1 \in \Sigma^\star$ be two partial traces. We define the following lattice operators:

- $\pi_0 \preceq \pi_1$ if and only if π_0 is a subtrace of π_1,
- $\pi_0 \curlywedge \pi_1 = \pi$ such that $(\pi \preceq \pi_1) \wedge (\pi \preceq \pi_2)$ and $(\forall \pi' : (\pi' \preceq \pi_1) \wedge (\pi' \preceq \pi_2)).\pi' \preceq \pi$.

$\Sigma^\star$ equipped with the order relation "$\preceq$" and the meet operator "$\curlywedge$", forms the meet semi lattice $\langle \Sigma^\star, \preceq, \curlywedge\rangle$.

This partial trace semantics can be expressed in a fixpoint form as well.

$$\Sigma^\star = \mathrm{lfp}^{\subseteq} F :$$
$$F \in \Sigma^\star \rightarrow \Sigma^\star$$
$$F(\mathsf{X}) \stackrel{\mathrm{def}}{=} \{\sigma \xrightarrow{\mathsf{a}'} \sigma' \in \mathsf{T} \mid \sigma \in \mathsf{I}\} \cup$$
$$\{\sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \mid \sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-2}} \sigma_{n-1} \in \mathsf{X} \wedge \sigma_{n-1} \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \in \mathsf{T}\}$$

Let $\langle \wp(\Sigma^\star), \subseteq, \emptyset, \Sigma^\star, \cap, \cup \rangle$ be a complete lattice of partial execution traces, where "$\subseteq$" is the classical subset relation, "$\cup$" is the set union and "$\cap$" the set intersection.

2.3 Abstract Domain: Pos

Among all the abstract domains which are used in abstract interpretation of logic programs, Pos has received considerable attention [2,11]. This domain is most commonly applied to the analysis of groundness dependencies for logic programs.

Let $\overline{\mathsf{V}} = \{\bar{\mathsf{x}}, \bar{\mathsf{y}}, \bar{\mathsf{z}}, \cdots\}$ be a countably infinite set of propositional variables and let $FP(\overline{\mathsf{V}})$ be the set of all finite subsets of variables of $\overline{\mathsf{V}}$. The set of propositional formulae containing variables in $\overline{\mathsf{V}}$ and logical connectives in $\Gamma \subseteq \{\wedge, \vee, \rightarrow, \neg\}$ is denoted by $\Omega(\Gamma)$. Similarly, given $\mathsf{U} \in FP(\overline{\mathsf{V}})$, the set of propositional formulae containing variables in U and connectives in Γ is denoted by $\Omega_{\mathsf{U}}(\Gamma)$.

A *truth-assignment* is a function $\overline{\Upsilon} : \overline{\mathsf{V}} \rightarrow \{\mathtt{T},\ \mathtt{F}\}$ that assigns to each propositional variable the value true ($\mathtt{T}$) or false ($\mathtt{F}$). Given a formula $\mathsf{f} \in \Omega(\Gamma)$, $\overline{\Upsilon} \models \mathsf{f}$ means that $\overline{\Upsilon}$ satisfies f, and $\mathsf{f}_1 \models \mathsf{f}_2$ is a shorthand for "$\overline{\Upsilon} \models \mathsf{f}_1$ implies $\overline{\Upsilon} \models \mathsf{f}_2$". $\Omega(\Gamma)$ is ordered by $\mathsf{f}_1 \trianglelefteq \mathsf{f}_2 \Leftrightarrow \mathsf{f}_1 \models \mathsf{f}_2$. Two formulae f_1 and f_2 are logically equivalent, denoted $\mathsf{f}_1 \equiv \mathsf{f}_2$ iff $\mathsf{f}_1 \trianglelefteq \mathsf{f}_2$ and $\mathsf{f}_2 \trianglelefteq \mathsf{f}_1$.

The *unit assignment* u is defined by $u(\mathsf{x}) = \mathtt{T}$ for all $\bar{\mathsf{x}} \in \overline{\mathsf{V}}$. We define the set of positive formulae by $\mathsf{Pos} = \{\mathsf{f} \in \Omega(\Gamma) \mid u \models \mathsf{f}\}$. Some obvious examples are $\mathtt{T}, \mathsf{x}_1 \in \mathsf{Pos}$ and $\mathtt{F}, \neg\mathsf{x}_1 \notin \mathsf{Pos}$.

We can consider the propositional formula ψ as a conjunction of subformulae $(\zeta_0 \wedge \ldots \wedge \zeta_n)$. We denote the set of subformulae of ψ as Sub_ψ. Let $\triangledown$ be the least upper bound operator on propositional formula defined by $\triangledown\{\psi_0, \ldots, \psi_n\} = \bigwedge\{Sub_{\psi_0}, \ldots, Sub_{\psi_n}\}$. $(\mathsf{Pos}, \trianglelefteq, \triangledown)$ forms a join semi lattice. Moreover, let $\ominus : \mathsf{Pos} \times \mathsf{Pos} \rightarrow \mathsf{Pos}$ be a binary operator defined as "simplification" between two propositional formulae: $\psi_0 \ominus \psi_1 = \bigwedge(Sub_{\psi_0} \setminus Sub_{\psi_1})$. This "simplification" permits us to obtain all the implication in ψ_0 which are not contained in ψ_1.

2.4 Abstract Semantics

Our approach is based on the abstract domain of logic formulae representing dependency between variables (which tracks the propagation of sensitive/insensitive information). The detection of possible information leakages is performed by evaluating formulae on truth-assignment functions. In particular, the analysis involves the following steps:

- Constructs at each program point the propositional formula ψ through a fixpoint algorithm which represents an over-approximation of variable's dependencies up to that program point.
- Partitions the variables into public and private privacy levels. Apply a truth-assignment function $\overline{\Upsilon}$ that assigns to each propositional variable the value T (true) or the value F (false) if the corresponding variable is private or public,

respectively. If $\overline{\Upsilon}$ does not satisfy ψ at all program points, then there could be some information leakages.

The logic formulae, obtained from program's instructions, are in the form:

$$\bigwedge_{0 \leq i \leq n \;\; 0 \leq j \leq m} \{\bar{\mathsf{x}}_{\mathsf{i}} \rightarrow \bar{\mathsf{y}}_{\mathsf{j}}\}$$

which means that the values of variable $\bar{\mathsf{y}}_{\mathsf{j}}$ could depend on the values of variable $\bar{\mathsf{x}}_{\mathsf{i}}$. For instance, the formula $\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}}$ represents variable dependency in assignment statement $\mathsf{x} := \mathsf{y}$; similarly, in case of conditional statement $\mathsf{if}(\mathsf{x} == 0)$ then $\mathsf{y} := \mathsf{z}$ we obtain the formula $(\bar{\mathsf{x}} \rightarrow \bar{\mathsf{y}}) \wedge (\bar{\mathsf{z}} \rightarrow \bar{\mathsf{y}})$. Notice that the propositional variable $\bar{\mathsf{v}}$ corresponds to the program variable v.

Formally, an abstract state $\sigma^\sharp \in \Sigma^\sharp \stackrel{\text{def}}{=} \mathsf{L} \times \mathsf{Pos}$ is a pair $\langle \ell, \phi \rangle$ where $\phi \in \mathsf{Pos}$ represents the dependencies occurred among program variables up to label $\ell \in \mathsf{L}$. Given a pair $\sigma^\sharp = \langle \ell, \phi \rangle$, we define $l(\sigma^\sharp) = \ell$ and $r(\sigma^\sharp) = \phi$. Let $BV(\mathsf{c})$, defined in Table 9, be the set of bound variables in command c.

Table 9. BV function

$$\begin{array}{rcl}
BV({}^{\ell}\mathsf{skip}) &=& \{\emptyset\} \\
BV({}^{\ell}\mathsf{v} := \mathsf{exp}) &=& \{\bar{\mathsf{v}}\} \\
BV(\mathsf{c}_0; \mathsf{c}_1) &=& BV(\mathsf{c}_0) \cup BV(\mathsf{c}_1) \\
BV(\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}) &=& BV(\mathsf{c}_0) \cup BV(\mathsf{c}_1) \\
BV(\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}) &=& BV(\mathsf{c})
\end{array}$$

Table 10. Abstract semantics

$$\begin{array}{l}
\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{skip}^{\ell_f}]\!] = \{\langle \ell, \psi \rangle \rightarrow \langle fin[\![{}^{\ell}\mathsf{skip}^{\ell_f}]\!], \psi \rangle\} \\
\overline{\mathsf{T}}[\![{}^{\ell}\mathsf{v} := \mathsf{exp}^{\ell_f}]\!] = \{\langle \ell, \psi \rangle \rightarrow \langle fin[\![{}^{\ell}\mathsf{v} := \mathsf{exp}^{\ell_f}]\!], \psi_0 \rangle\} \\
\overline{\mathsf{T}}[\![\mathsf{c}_0; \mathsf{c}_1]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!] \\
\overline{\mathsf{T}}[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}^{\ell_f}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}_0]\!] \cup \overline{\mathsf{T}}[\![\mathsf{c}_1]\!] \cup \\
\qquad\qquad \{\langle \ell, \psi \rangle \rightarrow \langle in[\![\mathsf{c}_0]\!], \psi \rangle\} \cup \{\langle \ell, \psi \rangle \rightarrow \langle in[\![\mathsf{c}_1]\!], \psi \rangle\} \cup \\
\qquad\qquad \{\langle \ell', \psi \rangle \rightarrow \langle fin[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}^{\ell_f}]\!], \psi_1 \rangle\} \\
\qquad\qquad \{\langle \ell', \psi \rangle \rightarrow \langle fin[\![\mathsf{if}\ {}^{\ell}\mathsf{b}\ \mathsf{then}\ \mathsf{c}_0\ \mathsf{else}\ \mathsf{c}_1\ {}^{\ell'}\mathsf{endif}^{\ell_f}]\!], \psi_2 \rangle\} \\
\overline{\mathsf{T}}[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}^{\ell_f}]\!] = \overline{\mathsf{T}}[\![\mathsf{c}]\!] \cup \{\langle \ell, \psi \rangle \rightarrow \langle in[\![\mathsf{c}]\!], \psi \rangle\} \cup \\
\qquad\qquad \{\langle \ell', \psi \rangle \rightarrow \langle fin[\![\mathsf{while}\ {}^{\ell}\mathsf{b}\ \mathsf{do}\ \mathsf{c}\ {}^{\ell'}\mathsf{done}^{\ell_f}]\!], \psi_3 \rangle\} \\
\\
\text{where} \\
\\
\psi_0 = \bigwedge\{\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}} \mid \bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{exp}) \wedge \bar{\mathsf{y}} \neq \bar{\mathsf{x}}\} \\
\qquad\qquad \bigwedge\{\bar{\mathsf{z}} \rightarrow \bar{\mathsf{w}} \mid \bar{\mathsf{z}} \rightarrow \bar{\mathsf{x}}, \bar{\mathsf{x}} \rightarrow \bar{\mathsf{w}} \in \psi\} \wedge (\psi \ominus \bigwedge\{\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}} \mid \bar{\mathsf{y}} \in \overline{\mathsf{V}} \wedge \bar{\mathsf{x}} \notin \overline{\mathsf{V}}[\![\mathsf{exp}]\!]\}) \\
\psi_1 = \bigwedge\{\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}} \mid \bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{b}) \wedge \bar{\mathsf{x}} \in BV(\mathsf{c}_0) \wedge \bar{\mathsf{y}} \neq \bar{\mathsf{x}}\} \wedge \psi \\
\psi_2 = \bigwedge\{\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}} \mid \bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{b}) \wedge \bar{\mathsf{x}} \in BV(\mathsf{c}_1) \wedge \bar{\mathsf{y}} \neq \bar{\mathsf{x}}\} \wedge \psi \\
\psi_3 = \bigwedge\{\bar{\mathsf{y}} \rightarrow \bar{\mathsf{x}} \mid \bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{b}) \wedge \bar{\mathsf{x}} \in BV(\mathsf{c}) \wedge \bar{\mathsf{y}} \neq \bar{\mathsf{x}}\} \wedge \psi
\end{array}$$

The *abstract semantics* of a command c is defined by $\overline{\mathsf{T}}[\![\mathsf{c}]\!]$. Similar to the concrete domain, we denote the transition from $\sigma_1^\sharp$ to $\sigma_2^\sharp$ by $\sigma_1^\sharp \to \sigma_2^\sharp$. The abstract semantics in the domain of propositional formulae is defined in Table 10.

Consider two sets of abstract states S_1 and S_2 such that $\mathsf{S}_1 = \{\langle \ell_0^1, \psi_0^1\rangle, \ldots, \langle \ell_n^1, \psi_n^1\rangle\}$ and $\mathsf{S}_2 = \{\langle \ell_0^2, \psi_0^2\rangle, \ldots, \langle \ell_m^2, \psi_m^2\rangle\}$. The partial ordering is defined by $\mathsf{S}_1 \sqsubseteq^\sharp \mathsf{S}_2 \Leftrightarrow n \leq m | \forall i \in [0, n], \ell_i^1 = \ell_i^2 \wedge \forall i \in [0, n], \psi_i^1 \trianglelefteq \psi_i^2$. Let $\mathsf{S}_0, \ldots \mathsf{S}_\mathsf{n} \in \wp(\Sigma^\sharp)$ be sets of abstract states. $\langle \wp(\Sigma^\sharp), \sqsubseteq^\sharp\rangle$ forms a poset since it is reflexive, antisymmetric, and transitive by basic properties of logic implication. The join operator $\sqcup^\sharp$ is defined by:

$$\begin{aligned}\sqcup^\sharp\{\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}\} = & \bigcup(\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}) \\ & \cup \{\langle \ell, \psi\rangle \mid \psi = \triangledown\{\psi' \mid \langle \ell, \psi'\rangle \in \bigcup(\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n})\}\} \\ & \setminus \{\langle \ell, \psi\rangle \in \bigcup(\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}) \mid \exists \langle \ell, \psi'\rangle \in \bigcup(\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}) \wedge \psi \neq \psi'\}\end{aligned}$$

and the meet operator $\sqcap^\sharp$ by:

$$\begin{aligned}\sqcap^\sharp\{\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}\} = & \{\langle \ell, \psi\rangle \in \mathsf{S}' \mid \mathsf{S}' \in \{\mathsf{S}_0, \ldots, \mathsf{S}_\mathsf{n}\} \wedge \\ & \forall i \in [0, n]. \exists \langle \ell, \psi_i'\rangle \in \mathsf{S}_\mathsf{i} \wedge \psi \trianglelefteq \psi_i'\}\end{aligned}$$

Basically, the join operator consists in the union of all elements. When two elements have the same label but different formula, the join operator takes the biggest one. Instead, the meet operator considers only the abstract states, with the same label, which are in all elements. In case of different formulae, the meet operator takes the smallest one. By definition join and meet operator are defined for every subset of elements of our domain. Therefore, we can conclude that $\langle \wp(\Sigma^\sharp), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp\rangle$ forms a complete lattice.

Let $\mathsf{I}^\sharp[\![\mathcal{P}]\!] = \{\langle in[\![\mathcal{P}]\!], \mathtt{T}\rangle\}$ be the set of possible initial abstract state of program $\mathcal{P}$. We define the *abstract semantics* as the set of all finite sets of abstract states, denoted by $\Sigma^{\star\sharp}$, reachable during one or more executions, in a finite time. For each element $\mathsf{S} \in \Sigma^{\star\sharp}$ we can denote by $\mathsf{S}^\dashv$ the set of terminal states, defined as $\mathsf{S}^\dashv = \{\sigma_0^\sharp \mid \nexists \sigma_1^\sharp \in \mathsf{S}. \sigma_0^\sharp \to \sigma_1^\sharp \in \overline{\mathsf{T}}\}$ and by $\ell(\mathsf{S})$ all labels of S. Let $\mathsf{S}_{\sigma_0^\sharp, \sigma_n^\sharp}$ denote a set of states, called *abstract sequence*, that contains a starting state $\sigma_0^\sharp$ and an ending state $\sigma_n^\sharp$ such that contains one or more traces from $\sigma_0^\sharp$ to $\sigma_n^\sharp$. We have that $\mathsf{S}^\dashv_{\sigma_0^\sharp, \sigma_n^\sharp} = \{\sigma_n^\sharp\}$.

We express the abstract semantics in a fixpoint form.

$$\begin{aligned}\Sigma^{\star\sharp} = & lfp^{\sqsubseteq} F^\sharp \text{ where } F^\sharp \in \Sigma^{\star\sharp} \to \Sigma^{\star\sharp} \\ F^\sharp(\mathsf{X}) \stackrel{\text{def}}{=} & \{\sigma^\sharp \mid \sigma^\sharp \in \mathsf{I}^\sharp\} \cup \{\mathsf{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \mid n \geq 1 \wedge \sigma_0^\sharp \in \mathsf{I}^\sharp \wedge \mathsf{S}_{\sigma_0^\sharp, \sigma_{n-1}^\sharp} \in \mathsf{X} \\ & \wedge \sigma_{n-1}^\sharp \to \sigma_n^\sharp \in \overline{\mathsf{T}}\} \cup \{\sqcup^\sharp\{\mathsf{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \mid \mathsf{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \in \mathsf{X}\}\}\end{aligned}$$

Example 1. In order to better understand how our dependency analysis works, consider the code in Fig. 1 and the program points 4, 5, 8, 10, 12 and 14. When we apply the steps defined above we obtain the propositional formulae in Table 11.

Table 11. Results of the analysis by Pos domain

Label	Propositional formula
4	$\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}$
5	$\overline{\mathsf{p}} \rightarrow \overline{\mathsf{sum}}$
8	$(\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}) \wedge (\overline{\mathsf{p}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{sum}})$
10	$(\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}) \wedge (\overline{\mathsf{p}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{x}} \rightarrow \overline{\mathsf{sum}})$
12	$(\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}) \wedge (\overline{\mathsf{p}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{x}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{sum}})$
14	$(\overline{\mathsf{x}} \rightarrow \overline{\mathsf{y}}) \wedge (\overline{\mathsf{p}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{x}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{y}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{n}} \rightarrow \overline{\mathsf{sum}}) \wedge$ $(\overline{\mathsf{i}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{i}} \rightarrow \overline{\mathsf{n}}) \wedge (\overline{\mathsf{k}} \rightarrow \overline{\mathsf{sum}}) \wedge (\overline{\mathsf{k}} \rightarrow \overline{\mathsf{n}})$

Through our analysis we tracked all the relation between variables. Suppose that variables $\{\overline{\mathsf{x}}, \overline{\mathsf{p}}\}$ are private, while all other variables are public. Formally, the correspondent truth-assignment function is defined by $\overline{\Upsilon} = \{\overline{\mathsf{x}}, \overline{\mathsf{p}} \mapsto \mathtt{T}\} \cup \{\overline{\mathsf{v}} \mapsto \mathtt{F} : \overline{\mathsf{v}} \in \mathsf{V} \setminus \{\overline{\mathsf{x}}, \overline{\mathsf{p}}\}\}$. $\overline{\Upsilon}$ does not satisfy the propositional formulae since in all considered program points there are some public variables that depends on one or more private variables.

Notice that we detect several spurious relations, too. For instance, in contrast with the obtained result, the variable $\overline{\mathsf{sum}}$ does not depend on $\overline{\mathsf{n}}$. Indeed at the end of the both branches the variable $\overline{\mathsf{sum}}$ has always the same value. In Sect. 3 we will refine the results through the domains combination.

2.5 An Instrumented Concrete Domain

To simplify the proof that our concrete and abstract domains from a Galois connection, we introduce another domain, isomorphic to the concrete domain. Let $\sigma^\diamond \in \Sigma^\diamond = \mathsf{L} \times \mathsf{A}$ be the set of states of this intermediate domain. A pair $\langle \ell, \mathsf{a} \rangle \in \mathsf{L} \times \mathsf{A}$ represents an action a which occurs at program label ℓ. Consider the set $\Sigma^{\star\diamond}$ which contains all the possible traces of $\sigma^\diamond$ that can occur during a finite computation. Given $\Pi_0^\diamond, \Pi_1^\diamond \in \wp(\Sigma^{\star\diamond})$, we define that $\Pi_0^\diamond \subseteq \Pi_1^\diamond$ if and only if for each $\pi_0^\diamond \in \Pi_0^\diamond$ there exists a $\pi_1^\diamond \in \Pi_1^\diamond$ such that $\pi_0^\diamond \preceq^\diamond \pi_1^\diamond$. We have that $\pi_0^\diamond \preceq^\diamond \pi_1^\diamond$ if and only if $\pi_0^\diamond$ is a subsequence of $\pi_1^\diamond$. Therefore $\langle \wp(\Sigma^{\star\diamond}), \subseteq, \emptyset, \Sigma^{\star\diamond}, \cap, \cup \rangle$ forms a lattice. Moreover, we denote by $\pi^{\diamond\dashv}$ the last state of the sequence.
We can relate $\wp(\Sigma^\star)$ and $\wp(\Sigma^{\star\diamond})$ by an abstraction $\alpha^\diamond \in \wp(\Sigma^\star) \rightarrow \wp(\Sigma^{\star\diamond})$ and a concretization $\gamma^\diamond \in \wp(\Sigma^{\star\diamond}) \rightarrow \wp(\Sigma^\star)$ function.

Let $\mathsf{X} = \{\pi_0, \ldots, \pi_n\} \in \wp(\Sigma^\star)$ be a set of partial traces and let $\mathsf{Y} = \{\pi_0^\diamond, \ldots, \pi_n^\diamond\} \in \wp(\Sigma^{\star\diamond})$ be a set of sequences of $\sigma^\diamond$.

$$\begin{aligned}
\alpha^\diamond(\mathsf{X}) \equiv& \{\langle \ell_0, \mathsf{a}_0 \rangle \rightarrow \ldots \rightarrow \langle \ell_m, \mathsf{a}_m \rangle \mid \sigma_0 \xrightarrow{\ell_0\, \mathsf{a}_0} \ldots \xrightarrow{\ell_m\, \mathsf{a}_m} \sigma_{m+1} \in \mathsf{X}\} \\
\gamma^\diamond(\mathsf{Y}) \equiv& \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \mathsf{Y}\}
\end{aligned}$$

Lemma 1. $\wp(\Sigma^\star) \underset{\alpha^\diamond}{\overset{\gamma^\diamond}{\leftrightarrows}} \wp(\Sigma^{\star\diamond})$ *forms an isomorphism, that is,* $\gamma^\diamond \circ \alpha^\diamond = \alpha^\diamond \circ \gamma^\diamond = id$ *(where id is the identity function).*

Proof. We have to prove that $\gamma^\diamond \circ \alpha^\diamond = \alpha^\diamond \circ \gamma^\diamond = id$, where id is the identity function. Let X and Y be elements of $\wp(\Sigma^{\star\diamond})$ and $\wp(\Sigma^\star)$ respectively.

$$\begin{aligned}
\alpha^\diamond(\gamma^\diamond(\mathsf{X})) &= \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \sigma_0 \xrightarrow{\ell_0\,\mathsf{a}_0} \ldots \xrightarrow{\ell_m\,\mathsf{a}_m} \sigma_{m+1} \in \gamma^\diamond(\mathsf{X})\} \\
&\quad \text{by definition of } \alpha^\diamond \\
&= \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \sigma_0 \xrightarrow{\ell_0\,\mathsf{a}_0} \ldots \xrightarrow{\ell_m\,\mathsf{a}_m} \sigma_{m+1} \\
&\quad \in \{\pi \mid \alpha^\diamond(\{\pi\}) \subseteq X\}\}\text{by definition of } \gamma^\diamond \\
&= \{\langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \mid \langle \ell_0, \mathsf{a}_0\rangle \to \ldots \to \langle \ell_m, \mathsf{a}_m\rangle \in \mathsf{X}\} \\
&= \mathsf{X}
\end{aligned}$$

$$\begin{aligned}
\gamma^\diamond(\alpha^\diamond(\mathsf{Y})) &= \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \alpha^\diamond(\mathsf{Y})\} \\
&\quad \text{by definition of } \gamma^\diamond \\
&= \{\pi \in \wp(\Sigma^\star) \mid \alpha^\diamond(\{\pi\}) \subseteq \{\alpha^\diamond(\{\pi'\}) \mid \pi' \in \mathsf{Y}\}\} \\
&\quad \text{by definition of } \alpha^\diamond \\
&= \{\pi \in \wp(\Sigma^\star) \mid \pi \in \mathsf{Y}\} \\
&= \mathsf{Y}
\end{aligned}$$

$\square$

Now we define the relation between $\wp(\Sigma^{\star\diamond})$ and $\wp(\Sigma^{\star\sharp})$ by $\alpha^\sharp$ and $\gamma^\sharp$. $\alpha^\sharp : \wp(\Sigma^{\star\diamond}) \to \wp(\Sigma^{\star\sharp})$ is defined by $\alpha^\sharp(\mathsf{X}) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \mathsf{X}\}$, where $\theta : \Sigma^{\star\diamond} \to \wp(\Sigma^{\star\sharp})$ is defined as follows.

$$\begin{aligned}
\theta(\mathsf{X}) = &\{\langle \ell, \psi\rangle \mid \forall \pi \in \mathsf{X}.\forall \pi' = \langle \ell_0, \mathsf{a}_0\rangle \to \langle \ell_m, \mathsf{a}_m\rangle \preceq^\diamond \pi : \\
&m \geq 0 \wedge \ell = \ell_m \wedge \psi = \mathsf{f}_0 \wedge \ldots \wedge \mathsf{f}_n\}
\end{aligned}$$

such that:

1. $(\forall \langle \ell, \mathsf{v} := \mathsf{exp}\rangle \in \pi' : \forall \langle \ell', \mathsf{v} := \mathsf{exp}'\rangle \in \pi'.\ell' \leq \ell).\exists \mathsf{f}_i = \bar{\mathsf{y}} \to \bar{\mathsf{v}} : \bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{exp})$
2. $\forall((\langle \ell_i, \mathsf{b}\rangle \to \ldots \to \langle \ell_j, \mathsf{endif}\rangle) \vee (\langle \ell_i, \mathsf{not\ b}\rangle \to \ldots \to \langle \ell_j, \mathsf{endif}\rangle)) \preceq^\diamond \pi^\diamond$ which represents an if statement and $\forall \langle \ell_k, \mathsf{v} := \mathsf{exp_k}\rangle : i < k < j$ exists $\mathsf{f}_h = \bar{\mathsf{y}} \to \bar{\mathsf{v}}$ such that $\bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{b})$.
3. $\forall((\langle \ell_i, \mathsf{b}\rangle \to \ldots \to \langle \ell_j, \mathsf{done}\rangle) \vee (\langle \ell_i, \mathsf{not\ b}\rangle \to \ldots \to \langle \ell_j, \mathsf{done}\rangle)) \preceq^\diamond \pi^\diamond$ which represents a while statement and $\forall \langle \ell_k, \mathsf{v} := \mathsf{exp}_k\rangle : i < k < j$ exists $\mathsf{f}_h = \bar{\mathsf{y}} \to \bar{\mathsf{v}}$ such that $\bar{\mathsf{y}} \in \overline{\mathsf{V}}(\mathsf{b})$.

Intuitively, the function θ transforms each action (or sequence of actions) in one or more propositional formulae. The easiest (case 1) applies when the action is an assignment statement ($\mathsf{v} := \mathsf{exp}$): we simply obtain the corresponding formula as defined in the transition semantics T. Instead, for if statements (case 2), we track all the assignment actions that are between if and endif. while statements are treated in a similar way (case 3).

Notice that $\langle \ell_i, \mathsf{b}\rangle \to \ldots \to \langle \ell_j, \mathsf{endif}\rangle$ (or $\langle \ell_i, \mathsf{not\ b}\rangle \to \ldots \to \langle \ell_j, \mathsf{endif}\rangle$) represents an if statement if and only if $\forall(\langle \ell_p, \mathsf{b}\rangle \vee \langle \ell_p, \mathsf{not\ b}\rangle) : i < p < j.\exists(\langle \ell_q, \mathsf{endif}\rangle \vee \langle \ell_q, \mathsf{done}\rangle) : p < q < j$ and $\forall(\langle \ell_q, \mathsf{endif}\rangle \vee \langle \ell_q, \mathsf{done}\rangle) : i < q < j.\exists(\langle \ell_p, \mathsf{b}\rangle \vee \langle \ell_p, \mathsf{not\ b}\rangle) : i < p < q$. Similarly for while statement.

Informally, the pair if and endif (or while and done) is an if (while) statement if and only if between these two actions, there are only assignments or other pairs if-endif or while-done which correspond to nested if and while statements. To better understand, consider the sequence $\cdots\langle \ell_0, \mathsf{b_0}\rangle \rightarrow \langle \ell_1, \mathsf{b_1}\rangle \rightarrow \langle \ell_2, \mathsf{v} := \mathsf{exp}\rangle \rightarrow \langle \ell_3, \mathsf{endif}\rangle \cdots$: the pairs $\langle \ell_0, \mathsf{b_0}\rangle$ and $\langle \ell_3, \mathsf{endif}\rangle$ are not an if statement because between these two actions there is $\langle \ell_1, \mathsf{b_1}\rangle$, which does not represent an assignment action neither an if statement.

The concretization function $\gamma^\sharp : \wp(\Sigma^{\star\sharp}) \rightarrow \wp(\Sigma^{\star\diamond})$ is defined by $\gamma^\sharp(\mathsf{Y}) = \{\pi^\diamond \in \Sigma^{\star\diamond} \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \mathsf{Y} \wedge l(\pi^{\diamond\dashv}) \in \ell(\mathsf{Y}^\dashv)\}$ where $\mathsf{Y} \in \wp(\Sigma^{\star\sharp})$.

Lemma 2. *$\theta : \Sigma^{\star\diamond} \rightarrow \wp(\Sigma^{\star\sharp})$ is monotonic:* $\mathsf{x} \preceq^\diamond \mathsf{y} \Rightarrow \theta(\mathsf{x}) \sqsubseteq^\sharp \theta(\mathsf{y})$

Proof. Let $\mathsf{x_0} = \{\sigma_0 \rightarrow \ldots \rightarrow \sigma_n\}$ and $\mathsf{x_1} = \{\sigma'_0 \rightarrow \ldots \rightarrow \sigma'_m\}$ be two elements of $\Sigma^{\star\diamond}$ such that $\mathsf{x_0} \preceq^\diamond \mathsf{x_1}$ and consider $\theta(\mathsf{x_0}) = \{{\sigma^\sharp}_0, \ldots, {\sigma^\sharp}_n\}$ and $\theta(\mathsf{x_1}) = \{{\sigma^\sharp}'_0, \ldots, {\sigma^\sharp}'_m\}$.

By the definition of "$\preceq^\diamond$" we know that $n \leq m$, $\forall i \in [0, n].\sigma_i = \sigma'_i$. Therefore, by the definition of θ, we have that $\forall i \in [0, n].{\sigma^\sharp}_i = {\sigma^\sharp}'_i$. Then, by definition of "$\sqsubseteq^\sharp$", $\theta(\mathsf{x_0}) \sqsubseteq^\sharp \theta(\mathsf{x_1})$. □

Lemma 3. *$\alpha^\sharp : \wp(\Sigma^{\star\diamond}) \rightarrow \wp(\Sigma^{\star\sharp})$ is monotonic:* $\mathsf{X} \subseteq \mathsf{Y} \Rightarrow \alpha^\sharp(\mathsf{X}) \sqsubseteq^\sharp \alpha^\sharp(\mathsf{Y})$

Proof. Consider $\mathsf{X_0}, \mathsf{X_1} \in \wp(\Sigma^{\star\diamond})$ such that $\mathsf{X_0} \subseteq \mathsf{X_1}$, $\alpha^\sharp(\mathsf{X_0}) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \mathsf{X_0}\}$ and $\alpha^\sharp(\mathsf{X_1}) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \mathsf{X_1}\}$. By definition of "$\subseteq$", $\forall \pi^\diamond \in \mathsf{X_0}, \exists \pi^\diamond \in \mathsf{X_1}$. By Lemma 2, $\theta(\pi^\diamond_0) \sqsubseteq^\sharp \theta(\pi^\diamond_1)$ for all $\pi^\diamond_0 \in \mathsf{X_0}$ and $\pi^\diamond_1 \in \mathsf{X_1}$. Then we have $\alpha^\sharp(\mathsf{X_0}) \sqsubseteq^\sharp \alpha^\sharp(\mathsf{X_1})$: $\alpha^\sharp(\mathsf{X_1})$ contains all the elements in $\alpha^\sharp(\mathsf{X_0})$. □

Lemma 4. *$\gamma^\sharp : \wp(\Sigma^{\star\sharp}) \rightarrow \wp(\Sigma^{\star\diamond})$ is monotonic:* $\mathsf{X} \sqsubseteq^\sharp \mathsf{Y} \Rightarrow \gamma^\sharp(\mathsf{X}) \subseteq \gamma^\sharp(\mathsf{Y})$

Proof. Consider $\mathsf{X_0}, \mathsf{X_1} \in \wp(\Sigma^{\star\sharp})$ such that $\mathsf{X_0} \sqsubseteq^\sharp \mathsf{X_1}$, $\gamma^\sharp(\mathsf{X_0}) = \{\pi^\diamond \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \mathsf{X_0} \wedge l(\pi^\diamond) \in \ell(\mathsf{X_0^\dashv})\}$ and $\gamma^\sharp(\mathsf{X_1}) = \{\pi^\diamond \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \mathsf{X_1} \wedge l(\pi^\diamond) \in \ell(\mathsf{X_1^\dashv})\}$. By definition of "$\sqsubseteq^\sharp$" and by Lemma 2, for all $\pi^\diamond_0 \in \gamma^\sharp(\mathsf{X_0})$ exists $\pi^\diamond_1 \in \gamma^\sharp(\mathsf{X_1})$. Therefore $\gamma^\sharp(\mathsf{X_0}) \sqsubseteq^\sharp \gamma^\sharp(\mathsf{X_1})$. □

Lemma 5. *$\alpha^\sharp \circ \gamma^\sharp$ is the identity:* $\alpha^\sharp(\gamma^\sharp(\mathsf{X})) = \mathsf{X}$

Proof. Let X be an element of $\wp(\Sigma^{\star\sharp})$. By definition of $\alpha^\sharp$, $\alpha^\sharp(\gamma^\sharp(\mathsf{X})) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \gamma^\sharp(\mathsf{X})\}$. By definition of $\gamma^\sharp$, $\alpha^\sharp(\gamma^\sharp(\mathsf{X})) = \sqcup^\sharp\{\theta(\pi^\diamond) \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \mathsf{X} \wedge l(\pi^\diamond) \in \ell(\mathsf{X}^\dashv)\}$. Then, $\alpha^\sharp(\gamma^\sharp(\mathsf{X}))$ contains the least upper bound of all the abstract traces that have the same last label of X and that are less or equal than X. Therefore $\alpha^\sharp(\gamma^\sharp(\mathsf{X})) = \mathsf{X}$. □

Lemma 6. *$\gamma^\sharp \circ \alpha^\sharp$ is extensive:* $\mathsf{X} \sqsubseteq^\sharp \gamma^\sharp(\alpha^\sharp(\mathsf{X}))$

Proof. Consider $\mathsf{X} \in \wp(\Sigma^{\star\diamond})$. By definition of $\gamma^\sharp$, $\gamma^\sharp(\alpha^\sharp(\mathsf{X})) = \{\pi^\diamond \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \alpha^\sharp(\mathsf{X}) \wedge l(\pi^\diamond) \in \ell(\alpha^\sharp(\mathsf{X})^\dashv)\}$. By definition of $\alpha^\sharp$, $\gamma^\sharp(\alpha^\sharp(\mathsf{X})) = \{\pi^\diamond \in \wp(\Sigma^{\star\diamond}) \mid \theta(\pi^\diamond) \sqsubseteq^\sharp \sqcup^\sharp\{\theta(\pi^\diamond) \mid \pi^\diamond \in \mathsf{X}\} \wedge l(\pi^\diamond) \in \ell(\alpha^\sharp(\mathsf{X})^\dashv)\}$. By definition of "$\sqcup^\sharp$", "$\sqsubseteq^\sharp$" and by Lemma 2, $\mathsf{X} \sqsubseteq^\sharp \gamma^\sharp(\alpha^\sharp(\mathsf{X}))$. □

Lemma 7. *$\wp(\Sigma^{\star\diamond}) \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion.*

Proof. $\wp(\Sigma^{\star\diamond})$ and $\wp(\Sigma^{\star\sharp})$ are two complete lattices, $\gamma^\sharp$ and $\alpha^\sharp$ are monotonic (Lemmas 3 and 4), $\alpha^\sharp \circ \gamma^\sharp$ is the identity (Lemma 5) and $\gamma^\sharp \circ \alpha^\sharp$ is extensive (Lemma 6). Therefore $\wp(\Sigma^{\star\diamond}) \underset{\alpha^\sharp}{\overset{\gamma^\sharp}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion. $\square$

Finally, we can express the relation between $\wp(\Sigma^\star)$ and $\wp(\Sigma^{\star\sharp})$ by the composition of above functions, $\alpha = \alpha^\sharp \circ \alpha^\diamond$ and $\gamma = \gamma^\diamond \circ \gamma^\sharp$.

Since the composition of an isomorphism and a Galois insertion is a Galois insertion, we can assert that $\wp(\Sigma^\star) \underset{\gamma^\sharp}{\overset{\alpha^\sharp}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ is a Galois insertion.

2.6 Properties

The aim of information flow analysis is to verify the confidentiality and the integrity of the information in computer programs. An information flow analysis can be carried out by considering different attacker abilities. In this context we consider two different scenarios: when the attacker can read public variables only at the beginning and at the end of the computation, and when the attacker can read public variables after each step of the computation. Note that the attacker, in both cases, knows the source code of the program.

Both the properties and the types of attacker are checked through the definition and the satisfiability of the propositional formulae (Pos) with respect to the truth-assignment function. Let $\overline{\Upsilon_\mathcal{P}} : \overline{\mathsf{V}} \rightarrow \{\mathtt{T}, \mathtt{F}\}$ be a truth-assignment function associated with the program $\mathcal{P}$. The security properties are modeled by the function definition, while the attacker is modeled by the set of propositional formulae we consider for the satisfiability. For the first case, in which the attacker can read public variables only at the beginning and at the end of the computation, the set of states to consider involves only the terminal states of each sequence ($\{\mathsf{S} \in \Sigma^{\star\sharp} \mid \overline{\Upsilon_\mathcal{P}} \vDash r(\mathsf{S}^{\dashv})\}$). Whereas in the second case, when the attacker can read public variables at each step of the computation, the set of states to consider involves all the propositional formulae in the sequence ($\{\mathsf{S} \in \Sigma^{\star\sharp} \mid \forall \sigma^\sharp \in \mathsf{S} : \overline{\Upsilon_\mathcal{P}} \vDash r(\sigma^\sharp)\}$).

Confidentiality. Confidentiality refers to limiting information access and disclosure to authorized users. For example, we require when we buy something online that our private data (e.g., credit card number) can be read only by the merchant.

Let $\Upsilon_\mathcal{P} : \mathsf{V} \rightarrow \{\mathsf{L}, \mathsf{H}\}$ be a function which assigns to each variable of program $\mathcal{P}$ a security class. $\mathcal{P}$ respects the confidentiality property, if and only if it does not contain any information leakage with respect to the function Υ_P, i.e., there is no information that moves from private to public variables. To verify this property, we define the corresponding truth-assignment function $\overline{\Upsilon_\mathcal{P}}$ as follows.

$$\overline{\Upsilon_\mathcal{P}}(\overline{\mathsf{x}}) = \begin{cases} \mathtt{T} & \text{if } \Upsilon_\mathcal{P}(\mathsf{x}) = \mathsf{H} \\ \mathtt{F} & \text{if } \Upsilon_\mathcal{P}(\mathsf{x}) = \mathsf{L} \end{cases}$$

Integrity. By integrity we mean that unauthorized people cannot modify a message.

Let $\Upsilon_{\mathcal{P}} : \mathsf{V} \rightarrow \{\mathsf{L}, \mathsf{H}\}$ be a function which assigns to each variable of program $\mathcal{P}$ a security class. The integrity property is verified if and only if public variables do not modify private variables, i.e., there is no information leakage from public variables to private variables. The corresponding truth-assignment function $\overline{\Upsilon_{\mathcal{P}}}$, to check this property, is defined as follows.

$$\overline{\Upsilon_{\mathcal{P}}}(\bar{\mathsf{x}}) = \begin{cases} \mathtt{T} & \text{if } \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{L} \\ \mathtt{F} & \text{if } \Upsilon_{\mathcal{P}}(\mathsf{x}) = \mathsf{H} \end{cases}$$

Notice that it is exactly the opposite of the truth-assignment function for the confidentiality property.

3 Combination of Symbolic and Numerical Domains

In this Section, we combine the symbolic propositional formulae domain described above with a numerical domain through reduced product, yielding to a refinement of the results obtained by the dependency analysis. Our modular construction allows to tune efficiency and accuracy changing the numerical domain. For instance, if we use intervals, we will be less precise than by using polyhedra, but we will obtain a more efficient analysis.

Let us briefly recall the main features of some numerical domains already in the literature.

Intervals. Intervals approximate a set of integers by an interval enclosing all of them. Formally, a set $\mathsf{V} \subseteq \mathbb{Z}$ is approximated with $[\mathsf{a}, \mathsf{b}]$ where $\mathsf{a} = \mathit{min}\ \mathsf{V}$ and $\mathsf{b} = \mathit{max}\ \mathsf{V}$. If it is not possible to know precisely the upper and lower bound of a set of integers a and b are $-\infty$ and $+\infty$, respectively. This domain is a lattice, and the ordering operator $\sqsubseteq$ is such that $[\mathsf{a}, \mathsf{b}] \sqsubseteq [\mathsf{c}, \mathsf{d}]$ if and only if the interval $[\mathsf{a}, \mathsf{b}]$ is contained by $[\mathsf{c}, \mathsf{d}]$. Therefore the top element is the interval $[-\infty, +\infty]$ and the bottom element is an interval such that $\mathsf{a} > \mathsf{b}$. This lattice has infinite height and contains infinite ascending chains. So it needs a widening operator. Intervals scale up, but in some cases they are too rough.

Polyhedra. Convex polyhedra are regions of some n-dimensional space that are bounded by a finite set of hyperplanes. A convex polyhedron in $\mathbb{R}^n$ describes a relation between n quantities. P. Cousot and N. Halbwachs [15] applied the theory of abstract interpretation to the static determination of linear equalities and inequalities among program variables by introducing the use of convex polyhedra as an abstract domain.

We denote by $\mathbf{v} = (v_0, \ldots v_{n-1}) \in \mathbb{R}^n$ a n-tuple (vector) of real numbers; $\mathbf{v} \cdot \mathbf{w}$ denotes the scalar product of vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$; the vector $\mathbf{0} \in \mathbb{R}$ has all components equal to zero. Let $\mathbf{x}$ be a n-tuple of distinct variables. Then $\beta = (\mathbf{a} \cdot \mathbf{x} \bowtie b)$ denotes a linear constraint, for each vector $\mathbf{a} \in \mathbb{R}^n$, where $\mathbf{a} \neq \mathbf{0}$, $b \in \mathbb{R}$ and $\bowtie = \{=, \geq, >\}$. A linear inequality constraint β defines an affine half-space of $\mathbb{R}^n$, denoted by $con(\{\beta\})$.

A set $\mathcal{P} \in \mathbb{R}^n$ is a (convex) polyhedron if and only if $\mathcal{P}$ can be expressed as the intersection of a finite number of affine half-spaces of $\mathbb{R}^n$, i.e., as the solution of a finite set of linear inequality constraints. The set of all polyhedra on the vector space $\mathbb{R}^n$ is denoted as $\mathsf{P_n}$. Let $\langle \mathsf{P_n}, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap \rangle$ be a lattice of convex polyhedra, where "$\subseteq$" is the set-inclusion, the empty set and $\mathbb{R}^n$ as the bottom and top elements, respectively. The binary meet operation returns the greatest polyhedron smaller than or equal to the two arguments, correspond to set intersection, and "$\uplus$" is the binary join operation and returns the least polyhedron greater than or equal to the two arguments. This abstract domain has exponential complexity, and it does not scale up in practice.

For more details about polyhedra, many works in literature define abstract domains based on polyhedra as Galois connection [6] and implement this domain [5,27].

Octagons. A. Miné introduced Octagons [35] for static analysis by abstract interpretation. The author extended a former numerical domain based on Difference-Bound Matrices [34] and showed practical algorithms to represent and manipulate invariants of the form $\pm \mathsf{x} \pm \mathsf{y} \leq \mathsf{c}$ (where x and y are program variables and c is a real constant) efficiently. Such invariants describe sets of point that are special kind of polyhedra called *octagons* because they feature at most eight edges in a two dimensional space.

The set of invariants which the analysis discovers is a subset of the ones discovered by Polyhedra, but it is quite efficient. In fact, it infers the invariants with a $\mathcal{O}(n^2)$ worst case memory complexity per abstract state and a $\mathcal{O}(n^3)$ worst case time complexity per abstract operation, where n is the number of variables in the program.

3.1 The Reduced Product

The best way to combine the propositional formulae domain $\langle \wp(\Sigma^{\star\sharp}), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$ and a numerical domain $\langle \aleph, \sqsubseteq^\aleph, \perp^\aleph, \top^\aleph, \sqcup^\aleph, \sqcap^\aleph \rangle$ is by using the reduced product operator [14].

Let $\wp(\Sigma^\star) \underset{\gamma_0}{\overset{\alpha_0}{\leftrightarrows}} \wp(\Sigma^{\star\sharp})$ and $\wp(\Sigma^\star) \underset{\gamma_1}{\overset{\alpha_1}{\leftrightarrows}} \aleph$ be two Galois connections and let $\varrho : \wp(\Sigma^{\star\sharp}) \times \aleph \rightarrow \wp(\Sigma^{\star\sharp}) \times \aleph$ be a reduce operator defined as follows: let $\mathsf{X} \in \wp(\Sigma^{\star\sharp})$ be a set of partial traces, and $\mathfrak{N} \in \aleph$ an element of the numerical domain (a set of intervals, an octagon or a polyhedron). Notice that whatever domain you choose, $\mathfrak{N}$ can be seen as a set of relations among variables value. The reduce operator ϱ is defined as $\varrho(\langle \mathsf{X}, \mathfrak{N} \rangle) = \langle \mathsf{X}', \mathfrak{N} \rangle$ where

$$\begin{aligned} \mathsf{X}' = &\{{\sigma^\sharp}_{new} \mid \forall \sigma^\sharp \in \mathsf{X}. l({\sigma^\sharp}_{new}) = l(\sigma^\sharp) \\ &\wedge r({\sigma^\sharp}_{new}) = (r(\sigma^\sharp) \ominus \{\bar{\mathsf{x}} \rightarrow \bar{\mathsf{y}} \mid \mathsf{y} = \mathsf{z} \in \mathfrak{N}, \mathsf{z} \in \overline{\mathbb{V}} \cup \mathbb{Z} \wedge \mathsf{z} \neq \mathsf{x}\})\} \end{aligned}$$

```
foo(){
        ⁰n = 0; ¹x = 1; ²i = 0; ³y = x−1; ⁴sum = p;
        while (⁵i<= k) do
                if (⁶n%2 == 0) then
                        ⁷sum = y + p; ⁸n = n +1;
                else
                        ⁹sum = x +(p−1); ¹⁰n = n+3;
                ¹¹endif
                ¹²i = i+1;
        ¹³done
}¹⁴
```

Fig. 1. Reduced product example

The reduced operator is aimed at excluding pointless dependencies for all variables which have the same value during the execution, without loosing purposeful relations (by the condition "$\mathsf{x} \neq \mathsf{z}$"). The reduce operator removes from the propositional formulae, contained in X, the implications which have at the right side a variable that has a constant value. In fact if the variable has a constant value, it cannot depend on other variables.

Then, the reduced product $\mathsf{D}^\natural$ is defined as follows:

$$\mathsf{D}^\natural = \{\varrho(\langle \mathsf{X}, \mathfrak{N}\rangle) \mid \mathsf{X} \in \wp(\Sigma^{\star\sharp}), \mathfrak{N} \in \aleph\}$$

Consider $\mathsf{X}_0, \mathsf{X}_1 \in \wp(\Sigma^{\star\sharp}), \mathfrak{N}_0, \mathfrak{N}_1 \in \aleph$ and $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle, \langle \mathsf{X}_1, \mathfrak{N}_1\rangle \in \mathsf{D}^\natural$. Then $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle \sqsubseteq^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle$ if and only if $\mathsf{X}_0 \sqsubseteq^\sharp \mathsf{X}_1$ and $\mathfrak{N}_0 \sqsubseteq^\aleph \mathfrak{N}_1$. We define the least upper bound and greatest lower bound operator by $\langle X_0, \mathfrak{N}_0\rangle \sqcup^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle = \langle \mathsf{X}_0 \sqcup^\sharp \mathsf{X}_1, \mathfrak{N}_0 \sqcup^\aleph \mathfrak{N}_1\rangle$ and $\langle \mathsf{X}_0, \mathfrak{N}_0\rangle \sqcap^\natural \langle \mathsf{X}_1, \mathfrak{N}_1\rangle = \langle \mathsf{X}_0 \sqcap^\natural \mathsf{X}_1, \mathfrak{N}_0 \sqcap^\aleph \mathfrak{N}_1\rangle$, respectively. $\langle \mathsf{D}^\natural, \sqsubseteq^\natural, \emptyset, \varrho(\langle \Sigma^{\star\sharp}, \mathbb{R}^n\rangle), \sqcup^\natural, \sqcap^\natural\rangle$ forms a complete lattice. In order to better understand the improvements yielded by the combination of the two domains consider the following example.

Example 2. Consider the code we introduced in Fig. 1. We adopt polyhedra as numerical domain. Below we report the results of two analyses for some program points.

	Polyhedra
4	$n = 0; x - 1 = 0; i = 0; y = 0$
5	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; 3i - n \geq 0;$
8	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k \geq 0; 3i - n \geq 0;$
10	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k \geq 0; 3i - n \geq 0;$
12	$-p + sum = 0; y = 0; x - 1 = 0; -i + n - 1 \geq 0; -i + k \geq 0;$
	$i \geq 0; 3i - n + 3 \geq 0;$
14	$-p + sum = 0; y = 0; x - 1 = 0; -i + n \geq 0; -i + k - 1 \geq 0; 3i - n \geq 0;$

	Propositional formula
4	$x \rightarrow y$
5	$p \rightarrow sum$
8	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (y \rightarrow sum)$
10	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum)$
12	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum) \wedge (y \rightarrow sum)$
14	$(x \rightarrow y) \wedge (p \rightarrow sum) \wedge (x \rightarrow sum) \wedge (y \rightarrow sum) \wedge (n \rightarrow sum) \wedge$ $(i \rightarrow sum) \wedge (i \rightarrow n) \wedge (k \rightarrow sum) \wedge (k \rightarrow n)$

When we apply the reduce operator defined above we obtain the following propositional formulas:

4	T
5	$p \rightarrow sum$
8	$p \rightarrow sum$
10	$p \rightarrow sum$
12	$p \rightarrow sum$
14	$(p \rightarrow sum) \wedge (i \rightarrow n) \wedge (k \rightarrow n)$

By using the reduce operator we simplified the propositional formulas, removing some implications which could in fact generate false alarms when using the direct product of the domains instead of the reduced product. For instance, in Pos analysis we track the relation $y \rightarrow sum$. At the same time, in the numerical analysis, we detect that variable sum is always equal to p (namely it is constant). This means that $y \rightarrow sum$ is a false alarm, hence by the reduce product we may delete it. At the same time, we cannot remove the relation between sum and p because it is detected also by the numerical analysis.

4 An Extension to Database Query Languages

In this section, we extend the full power of the proposed model to the case of data-intensive applications embedding SQL statements, in order to identify possible leakage of sensitive database information as well. This is particular important as in fact unauthorized leakage often occurs while propagating through database applications accessing and processing them legitimately.

4.1 A Motivating Example

Consider the database of Table 12 where customer's personal information and journey-details are stored in tables "Customer" and "Travel" respectively. On booking a particular flight by a customer, the journey details are added to the table "Travel" and the source-destination distance is added to the corresponding entry in 'DistanceCovered' attribute of the table "Customer". Observe that 10 points on the journey each 100 Km are offered which is reflected in the attribute 'Points'. In addition, a boarding-priority value in the attribute 'BoardPriority' is assigned to each journey based on the points acquired by the passenger. This is depicted by procedure `BookFlight()` in program $\mathcal{P}$ in Fig. 2.

Table 12. Database dB

(a) Table "Customer"

custID	custName	Address	Age	DistanceCovered	Points
1	Alberto	Athens	56	650	60
2	Matteo	Venice	68	49	0
3	Francesco	Washington	38	972	90
4	Smith	Paris	42	185	10

(b) Table "Travel"

custID	Source	Destination	FlightID	JourneyDate	BoardPriority
1	A	B	F139	26-04-14	2
2	C	D	F28	16-11-13	0
3	A	B	F139	26-04-14	3
4	A	B	F139	26-04-14	1

```
Function BookFlight()
$flight=checkAvailability($source, $dest);
if($flight ≠ NULL){
 $dist=computeDistance($source, $dest);
 UPDATE Customer SET DistanceCovered = DistanceCovered +
 $dist WHERE custID=$id;
 UPDATE Customer SET Points = Points + 10 ×
 FLOOR($dist/100) WHERE custID=$id;
 ResultSet rs = SELECT Points FROM Customer WHERE
 custID=$id;
 while(rs.next()){
    $point=rs.next().Points;
    $priority=getPriority($point);
    INSERT INTO Travel(userID, Source, Destination,
    FlightID, JourneyDate, BoardPriority) VALUES
    ($id,$source,$dest,$flight,$date,$priority);}}
End of Function BookFlight()

 ⋮

Function Upgrade()
ResultSet rs = SELECT custID, DistanceCovered, Points FROM
Customer WHERE Points>50;
while(rs.next()){
 $id=rs.next().custID;
 $point=rs.next().Points;
 UPDATE Travel SET BoardPriority=BoardPriority +
 ($point-50)/10 WHERE custID=$id;}
End of Function Upgrade()
```

Fig. 2. Program $\mathcal{P}$

Assume that values of the attributes 'Address', 'Age', 'DistanceCovered' and 'Points' in table "Customer" are private, whereas the information in Table "Travel" is public. To distinguish from the database attributes, we prefix \$ to the application variables in $\mathcal{P}$. Finally, suppose the company has decided to upgrade the customers having more than 50 'Points' to the status of 'BoardPriority'. This is expressed in $\mathcal{P}$ by the activation of the `Upgrade()` function.

It is clear from the code that the values of 'BoardPriority' in tuples where 'custID' are equal to '1' and '3' will be upgraded from 2 to 3 and from 3 and 7 respectively. Therefore, an attacker can easily deduce the exact values of sensitive attribute 'Points' in Table "Customer", by observing the change that occurred in the public attribute 'BoardPriority' in Table "Travel".

The example above clearly shows that sensitive database information may be leaked through database applications when public attribute values depend, directly or indirectly, on private attribute values or private application variable values in the program. For instance, in the given example, the leakage occurs due to the dependence "Points→ BoardPriority" at program label 19.

4.2 Labeled Syntax and Concrete Semantics

The labeled syntax description of the language, depicted in Table 13, includes imperative statements embedding SQL. We express an SQL statement by a tuple $\langle \mathtt{OP}, \phi \rangle$, where ϕ is a precondition following first-order logic which is used to identify a set of tuples in the database on which the appropriate operation `OP` (either select, or insert, or update, or delete) is performed. Each operation represents a set of actions, *e.g.* select operation includes `GROUP BY`, aggregate functions, `ORDER BY`, etc. Observe that applications embedding SQL statements involve two distinct sets of variables: application variables $\mathsf{V_a}$ and database variables $\mathsf{V_d}$. Variables from $\mathsf{V_d}$ appear only in the SQL statements, whereas variables in $\mathsf{V_a}$ may appear in all types of instructions (either SQL or imperative).

We define the action function a and variable function V for the language in Tables 14 and 15 respectively.

Lets recall from [23] the notion of *environments* correspond to the variables in $\mathsf{V_a}$ and $\mathsf{V_d}$ respectively.

An *application environment* $\rho_a \in \mathcal{E}_a$ maps a variable $x \in dom(\rho_a) \subseteq \mathsf{V_a}$ to its value $\rho_a(x)$. So, $\mathcal{E}_a \triangleq \mathsf{V_a} \longmapsto \mathfrak{D}_{\mho}$ where $\mathfrak{D}_{\mho}$ is the semantic domain for $\mathsf{V_a}$.

Consider a database as a set of indexed tables $\{t_i \mid i \in I_x\}$ for a given set of indexes I_x. A *database environment* is defined by a function ρ_d whose domain is I_x, such that for $i \in I_x,\ \rho_d(i) = t_i$.

Given a database environment ρ_d and a table $t \in d$. Assume $attr(t) = \{a_1, a_2, ..., a_k\}$. So, $t \subseteq D_1 \times D_2 \times .... \times D_k$ where a_i is the attribute corresponding to the typed domain D_i. A *table environment* ρ_t for a table t is defined as a function such that for any attribute $a_i \in attr(t)$, $\rho_t(a_i) = \langle \pi_i(l_j) \mid l_j \in t \rangle$, where π is the projection operator and $\pi_i(l_j)$ represents i^{th} element of the l_j-th row. In other words, ρ_t maps a_i to the ordered set of values over the rows of the table t.

A *state* $\sigma \in \Sigma \triangleq \mathsf{L} \times \mathcal{E}_d \times \mathcal{E}_a$ is denoted by a tuple $\langle \ell, \rho_d, \rho_a \rangle$ where $\ell \in \mathsf{L}$, $\rho_d \in \mathcal{E}_d$ and $\rho_a \in \mathcal{E}_a$ are the label of the statement to be executed, the database environment and the application environment respectively.

The set of *states* of a program $\mathcal{P}$ is, thus, defined as $\Sigma[\![\mathcal{P}]\!] \triangleq \mathsf{L}[\![\mathcal{P}]\!] \times \mathcal{E}_d[\![\mathcal{P}]\!] \times \mathcal{E}_a[\![\mathcal{P}]\!]$, where $\mathsf{L}[\![\mathcal{P}]\!]$ is the set of labels in $\mathcal{P}$, and $\mathcal{E}_d[\![\mathcal{P}]\!]$ and $\mathcal{E}_a[\![\mathcal{P}]\!]$ are the sets

Table 13. Syntax of labeled programs embedding SQL

Constants		
$\mathsf{k} \in \mathsf{K}$		Set of Constants
$\mathsf{k} ::= n \mid s$		
	where $n \in \mathbb{N}$, $s \in$ Strings	
Variables		
$\mathsf{v_a} \in \mathsf{V_a}$		Set of Application Variables
$\mathsf{v_a} ::= x \mid y \mid z \mid \ldots$		
$\mathsf{v_d} \in \mathsf{V_d}$		Set of Database Attributes
$\mathsf{v_d} ::= a_1 \mid a_2 \mid a_3 \mid \ldots$		
Expressions		
$\mathsf{exp} \in \mathsf{E}$		Set of Arithmetic Expressions
$\mathsf{exp} ::= \mathsf{k} \mid \mathsf{v_d} \mid \mathsf{v_a} \mid \mathsf{exp_1} \oplus \mathsf{exp_2}$		
	where $\oplus \in \{+, -, *, /\}$	
$\mathsf{b} \in \mathsf{B}$		Set of Boolean Expressions
$\mathsf{b} ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{exp_1} \oslash \mathsf{exp_2} \mid \neg \mathsf{b} \mid \mathsf{b_1} \otimes \mathsf{b_2}$		
	where $\oslash \in \{\leq, \geq, ==, >, \neq, \ldots\}$ and $\otimes \in \{\vee, \wedge\}$	
SQL Preconditions		
$\tau \in \mathsf{T}$		Set of Terms
$\tau ::= \mathsf{k} \mid \mathsf{v_d} \mid \mathsf{v_a} \mid f_n(\tau_1, \tau_2, ..., \tau_n)$		
	where f_n is an n-ary function.	
$\mathsf{a_f} \in \mathsf{A_f}$		Set of Atomic Formulas
$\mathsf{a_f} ::= R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 == \tau_2$		
	where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{\mathsf{true}, \mathsf{false}\}$	
$\phi \in \mathsf{W}$		Set of Pre-conditions
$\phi ::= \mathsf{a_f} \mid \neg\phi \mid \phi_1 \otimes \phi_2 \mid \odot \mathsf{v}\, \phi$		
	where $\otimes \in \{\vee, \wedge\}$ and $\odot \in \{\forall, \exists\}$ and $\mathsf{v} \in (\mathsf{V_a} \cup \mathsf{V_d})$	
SQL Functions		
$g(\mathbf{exp}) ::= \mathtt{GROUP\ BY}(\mathbf{exp}) \mid id$		
	where $\mathbf{exp} = \langle \mathsf{exp_1}, ..., \mathsf{exp_n} \mid \mathsf{exp_i} \in \mathsf{E} \rangle$	
$e ::= \mathtt{DISTINCT} \mid \mathtt{ALL}$		
$s ::= \mathtt{AVG} \mid \mathtt{SUM} \mid \mathtt{MAX} \mid \mathtt{MIN} \mid \mathtt{COUNT}$		
$h(\mathsf{exp}) ::= s \circ e(\mathsf{exp}) \mid \mathtt{DISTINCT}(\mathsf{exp}) \mid id$		
$h(*) ::= \mathtt{COUNT(*)}$		
	where * represents a list of database attributes denoted by $\mathbf{v_d}$	
$\boldsymbol{h}(\boldsymbol{u}) ::= \langle h_1(u_1), ..., h_n(u_n) \rangle$		
	where $\boldsymbol{h} = \langle h_1, ..., h_n \rangle$ and $\boldsymbol{u} = \langle u_1, ..., u_n \mid u_i = \mathsf{exp} \vee u_i = * \rangle$	
$f(\mathbf{exp}) ::= \mathtt{ORDER\ BY\ ASC}(\mathbf{exp}) \mid \mathtt{ORDER\ BY\ DESC}(\mathbf{exp}) \mid id$		
Labeled Commands		
$\ell \in \mathsf{L}$		Set of Labels
$\mathsf{q} \in \mathsf{Q}$		Set of Labeled SQL Statements
$\mathsf{q} ::=$ SELECT | UPDATE | INSERT | DELETE		
SELECT $::= \langle {}^{\ell_5}assign(\mathsf{v_a}),\ {}^{\ell_4}f(\mathbf{exp'}),\ {}^{\ell_3}e(\boldsymbol{h}(\boldsymbol{u})),\ {}^{\ell_2}\phi',\ {}^{\ell_1}g(\mathbf{exp}),\ {}^{\ell_0}\phi \rangle$		
UPDATE $::= \langle {}^{\ell'}\mathbf{v_d} \stackrel{upd}{=} \mathbf{exp},\ {}^{\ell}\phi \rangle$		
INSERT $::= \langle {}^{\ell'}\mathbf{v_d} \stackrel{new}{=} \mathbf{exp},\ {}^{\ell}\mathsf{true} \rangle$		
DELETE $::= \langle {}^{\ell'}del(\mathbf{v_d}),\ {}^{\ell}\phi \rangle$		
$\mathsf{c} \in \mathsf{C}$		Set of Labeled Commands
$\mathsf{c} ::= {}^{\ell}skip \mid {}^{\ell}\mathsf{v_a} = \mathsf{exp} \mid \mathsf{q} \mid \mathsf{c_1}; \mathsf{c_2}$		
$\mid$ if ${}^{\ell}$b then $\mathsf{c_1}$ else $\mathsf{c_2}$ ${}^{\ell'}$endif		
$\mid$ while ${}^{\ell}$b do c ${}^{\ell'}$done		
$\mathcal{P} ::= \mathsf{c}^{\ell}$		Program that ends with label ℓ.

Table 14. Definition of action function a

$a[\![\text{SELECT}]\!] \stackrel{def}{=} \{^{\ell_5} assign(\mathsf{v_a}),\ ^{\ell_4} f(\mathbf{exp'}),\ ^{\ell_3} e(h(u)),\ ^{\ell_2}\phi',\ ^{\ell_1} g(\mathbf{exp}),\ ^{\ell_0}\phi\}$
$a[\![\text{UPDATE}]\!] \stackrel{def}{=} \{^{\ell'} \mathbf{v_d} \stackrel{upd}{=} \mathbf{exp},\ ^{\ell}\phi\}$
$a[\![\text{INSERT}]\!] \stackrel{def}{=} \{^{\ell'} \mathbf{v_d} \stackrel{new}{=} \mathbf{exp}\}$
$a[\![\text{DELETE}]\!] \stackrel{def}{=} \{^{\ell'} del(\mathbf{v_d}),\ ^{\ell}\phi\}$

Table 15. Definition of variables function V

$\mathsf{V}[\![\mathsf{c}]\!]$	$\stackrel{def}{=}$	$\emptyset$
$\mathsf{V}[\![\mathsf{v}]\!]$	$\stackrel{def}{=}$	$\{\mathsf{v}\}$, where $\mathsf{v} \in (\mathsf{V_a} \cup \mathsf{V_d})$
$\mathsf{V}[\![\mathbf{v}]\!]$	$\stackrel{def}{=}$	$\bigcup_{\mathsf{v_i}\in\mathbf{v}} \mathsf{V}[\![\mathsf{v_i}]\!]$
$\mathsf{V}[\![\mathsf{exp_1} \oplus \mathsf{exp_1}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathsf{exp_1}]\!] \cup \mathsf{V}[\![\mathsf{exp_1}]\!]$, where $\oplus \in \{+,-,*,/\}$
$\mathsf{V}[\![\mathbf{exp}]\!]$	$\stackrel{def}{=}$	$\bigcup_{\mathsf{exp_i}\in\mathbf{exp}} \mathsf{V}[\![\mathsf{exp_i}]\!]$
$\mathsf{V}[\![\mathsf{true}]\!]$	$\stackrel{def}{=}$	$\emptyset$
$\mathsf{V}[\![\mathsf{false}]\!]$	$\stackrel{def}{=}$	$\emptyset$
$\mathsf{V}[\![\mathsf{exp_1} \oslash \mathsf{exp_2}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathsf{exp_1}]\!] \cup \mathsf{V}[\![\mathsf{exp_2}]\!]$, where $\oslash \in \{\leq, \geq, ==, >, \neq, \dots\}$
$\mathsf{V}[\![\neg\mathsf{b}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathsf{b}]\!]$
$\mathsf{V}[\![\mathsf{b_1} \otimes \mathsf{b_2}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathsf{b_1}]\!] \cup \mathsf{V}[\![\mathsf{b_2}]\!]$, where $\otimes \in \{\vee, \wedge\}$
$\mathsf{V}[\![f_n(\tau_1,\dots,\tau_n)]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\tau_1]\!] \cup \dots \cup \mathsf{V}[\![\tau_n]\!]$, where f_n is an n-ary function.
$\mathsf{V}[\![R_n(\tau_1,\dots,\tau_n)]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\tau_1]\!] \cup \dots \cup \mathsf{V}[\![\tau_n]\!]$, where $R_n(\tau_1,\tau_2,...,\tau_n) \in \{\mathsf{true}, \mathsf{false}\}$
$\mathsf{V}[\![\tau_1 = \tau_2)]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\tau_1]\!] \cup \mathsf{V}[\![\tau_2]\!]$
$\mathsf{V}[\![\neg\phi]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\phi]\!]$
$\mathsf{V}[\![\phi_1 \otimes \phi_2]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\phi_1]\!] \cup \mathsf{V}[\![\phi_2]\!]$, where $\otimes \in \{\vee, \wedge\}$
$\mathsf{V}[\![\odot\mathsf{v}\ \phi]\!]$	$\stackrel{def}{=}$	$\{\mathsf{v}\} \cup \mathsf{V}[\![\phi]\!]$, where $\odot \in \{\forall, \exists\}$
$\mathsf{V}[\![\text{SELECT}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathbf{v_a}]\!] \cup \mathsf{V}[\![\mathbf{exp'}]\!] \cup \mathsf{V}[\![u]\!] \cup \mathsf{V}[\![\phi']\!] \cup \mathsf{V}[\![\mathbf{exp}]\!] \cup \mathsf{V}[\![\phi]\!]$
$\mathsf{V}[\![\text{UPDATE}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathbf{v_d}]\!] \cup \mathsf{V}[\![\mathbf{exp}]\!] \cup \mathsf{V}[\![\phi]\!]$
$\mathsf{V}[\![\text{INSERT}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathbf{v_d}]\!] \cup \mathsf{V}[\![\mathbf{exp}]\!]$
$\mathsf{V}[\![\text{DELETE}]\!]$	$\stackrel{def}{=}$	$\mathsf{V}[\![\mathbf{v_d}]\!] \cup \mathsf{V}[\![\phi]\!]$

of database and application environments whose domain is the set of database and application variables in $\mathcal{P}$ only.

The *labeled transition relation* $\mathsf{T} : \ \Sigma \times \mathsf{A} \longmapsto \wp(\Sigma)$ specifies which successor states $\sigma' = \langle \ell', \rho_{d'}, \rho_{a'} \rangle \in \Sigma$ can follow when an action $\mathsf{a} \in \mathsf{A}$ executes on state $\sigma = \langle \ell, \rho_d, \rho_a \rangle \in \Sigma$. We denote a labeled transition by $\sigma \xrightarrow{\mathsf{a}} \sigma'$ or by $\langle \ell, \rho_d, \rho_a \rangle \xrightarrow{\mathsf{a}} \langle \ell', \rho_{d'}, \rho_{a'} \rangle$, or by $\langle \ell, \rho \rangle \xrightarrow{\mathsf{a}} \langle \ell', \rho' \rangle$ where ρ and ρ' represent (ρ_d, ρ_a) and $(\rho_{d'}, \rho_{a'})$ respectively.

The *labeled transition semantics* $\mathsf{T}[\![\mathcal{P}]\!] \in \wp(\Sigma[\![\mathcal{P}]\!] \times a[\![\mathcal{P}]\!] \longmapsto \wp(\Sigma[\![\mathcal{P}]\!]))$ of a program $\mathcal{P}$ restricts the transition relation to program actions, *i.e.*

$$\mathsf{T}[\![\mathcal{P}]\!]\sigma = \{\sigma' \mid \sigma \xrightarrow{\mathsf{a}} \sigma' \wedge \mathsf{a} \in a[\![\mathcal{P}]\!] \wedge \sigma, \sigma' \in \Sigma[\![\mathcal{P}]\!]\}$$

The *labeled transition semantics* of various commands in database applications can easily be defined from the semantic description reported in [23].

Given a program $\mathcal{P}$, let $\mathsf{I} = \{\langle in[\![\mathcal{P}]\!], \rho_d, \rho_a\rangle \mid \rho_a \in \mathcal{E}_a \wedge \rho_d \in \mathcal{E}_d\}$ be the set of initial states of $\mathcal{P}$. The *partial trace semantics* of $\mathcal{P}$ can be defined as

$$\mathsf{T}[\![\mathcal{P}]\!](\mathsf{I}) = \mathrm{lfp}_{\emptyset}^{\subseteq} F(\mathsf{I}) = \bigcup_{i \leq \omega} F^i(\mathsf{I})$$

where $F(\mathsf{I}_0) = \lambda \mathsf{X}.\ \mathsf{I}_0 \cup \{\sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \xrightarrow{\mathsf{a}_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{\mathsf{a}_0} \dots \xrightarrow{\mathsf{a}_{n-1}} \sigma_n \in \mathsf{X}$
$\wedge \sigma_n \xrightarrow{\mathsf{a}_n} \sigma_{n+1} \in \mathsf{T}[\![\mathcal{P}]\!]\}$

4.3 Abstract Semantics

In case of applications embedding SQL statements, we need to consider two additional dependences, called *database-database* dependence and *program-database* dependence [24]. A *program-database* dependence arises between a database variable and an application variable, where values of the database variable depend on the value of the program variable or vice-versa. A *database-database* dependence arises between two database variables where the values of one depend on the values of the other.

Example 3. Consider the database of Table 12 in Sect. 4.1. Consider the following SELECT query:

$\mathsf{q}_1 =$ SELECT *Points*, AVG(*Age*) INTO $\mathsf{v_a}$ FROM *Customer* WHERE *Points* >=50 GROUP BY *Points* HAVING SUM(*DistanceCovered*)>100 ORDER BY *Points*

Note that we use "INTO $\mathsf{v_a}$" in q_1 to mention that the result of the query is finally assigned to $\mathsf{v_a}$, where $\mathsf{v_a}$ is a Record or ResultSet type application variable with fields $\boldsymbol{w} = \langle w_1, w_2\rangle$. The type of w_1, w_2 are same as the return type of '*Points*', 'AVG(*Age*)' respectively. Recall from Table 13 that the syntax of SELECT statement is defined as:

$$\langle {}^{\ell_5} assign(\mathsf{v_a}),\ {}^{\ell_4} f(\mathbf{exp'}),\ {}^{\ell_3} e(\boldsymbol{h}(\boldsymbol{u})),\ {}^{\ell_2}\phi',\ {}^{\ell_1} g(\mathbf{exp}),\ {}^{\ell_0}\phi\rangle$$

According the syntax defined above, q_1 can be formulated as:

$\mathsf{q}_1 =$ SELECT $e(\boldsymbol{h}(\boldsymbol{u}))$ INTO $\mathsf{v_a}(\boldsymbol{w})$ FROM *Customer* WHERE ϕ GROUP BY($\mathbf{exp}$) HAVING ϕ' ORDER BY ASC($\mathbf{exp'}$)

where

- $\phi = Points$ >=50
- $\mathbf{exp} = \langle Points\rangle$
- $g(\mathbf{exp}) =$ GROUP BY($\langle Points\rangle$)
- $\phi' =$ (SUM∘ALL(*DistanceCovered*))>100
- $\boldsymbol{h} = \langle$DISTINCT, AVG∘ALL$\rangle$
  $\boldsymbol{u} = \langle Points, Age\rangle$
  $\boldsymbol{h}(\boldsymbol{u}) = \langle$DISTINCT(*Points*), AVG∘ALL(*Age*)$\rangle$
- $\mathbf{exp'} = \langle Points\rangle$
- $f(\mathbf{exp'}) =$ ORDER BY ASC($\langle Points\rangle$)

– $\mathsf{v_a}$ = Record or ResultSet type application variable with fields $\boldsymbol{w}=\langle w_1, w_2\rangle$. The type of w_1 and w_2 are same as the return type of `DISTINCT`($Points$) and `AVG`∘`ALL`(Age) respectively.

From $\mathsf{q_1}$ we get the following set of logical formula representing variable dependences:

$$\overline{Points} \rightarrow \overline{\mathsf{v_a}.w_1}, \quad \overline{Age} \rightarrow \overline{\mathsf{v_a}.w_2}, \quad \overline{Points} \rightarrow \overline{\mathsf{v_a}.w_2}$$
$$\overline{DistanceCovered} \rightarrow \overline{\mathsf{v_a}.w_1}, \overline{DistanceCovered} \rightarrow \overline{\mathsf{v_a}.w_2}$$

Below we depict variable dependences in other SQL commands.

$\mathsf{q_2}$=`UPDATE` $Customer$ `SET` $DistanceCovered = \$y + 150$ `WHERE` $custID$=2
/* where $\$y$ is an application variable. */

The logical formula obtained from $\mathsf{q_2}$ are: $\overline{custID} \rightarrow \overline{DistanceCovered}$, $\overline{\$y} \rightarrow \overline{DistanceCovered}$.

$\mathsf{q_3}$ = `INSERT INTO` $Travel\big(custID, Source, Destination, FlightID, JourneyDate, BoardPriority\big)$ `VALUES` $\big(5, \text{"}D\text{"}, \text{"}E\text{"}, \text{"}F34\text{"}, \$y, \$z\big)$
/* where $\$y$ and $\$z$ are application variables. */

The logical formula obtained from $\mathsf{q_3}$ are: $\overline{\$y} \rightarrow \overline{JourneyDate}$, $\overline{\$z} \rightarrow \overline{BoardPriority}$.

$\mathsf{q_4}$ = `DELETE FROM` $Customer$ `WHERE` Age >60

The logical formula obtained from $\mathsf{q_4}$ are:

$$\overline{Age} \rightarrow \overline{custID}, \overline{Age} \rightarrow \overline{custName}, \quad \overline{Age} \rightarrow \overline{Address}$$
$$\overline{Age} \rightarrow \overline{Age}, \quad \overline{Age} \rightarrow \overline{DistanceCovered}, \overline{Age} \rightarrow \overline{Points}$$

The dependences above indicate explicit-flow of information. An example of implicit-flow that may occur in case of our application is, for instance, when manipulation of any public database information is performed under the control statements involving high variables.

Table 16 depicts abstract labeled transition semantics of various statements in database applications. The abstract semantics of the program is obtained by fix-point computation over the abstract domain.

4.4 Enhancing the Analysis

The dependences that we considered so far are syntax-based, and may yield false positives in the analysis. For instance, let us consider the database in Table 17 and the query $\mathsf{q_5}$.

$\mathsf{q_5}$ = `SELECT` $Type$ `INTO` $\mathsf{v_a}$ `FROM` Emp, Job `WHERE` $Sal = BASIC+(BASIC * (DA/100))+(BASIC * (HRA/100))$

Table 16. Definition of abstract transition function $\overline{\mathsf{T}}$

$$
\begin{aligned}
&\overline{\mathsf{T}}[\![\text{SELECT}]\!]\\
&\stackrel{def}{=} \overline{\mathsf{T}}[\![\langle {}^{\ell_5} assign(\mathsf{v_a}),\ {}^{\ell_4} f(\mathbf{exp'}),\ {}^{\ell_3} e(h(u)),\ {}^{\ell_2}\phi',\ {}^{\ell_1} g(\mathbf{exp}),\ {}^{\ell_0}\phi\rangle]\!]\\
&\stackrel{def}{=} \{\langle \ell_0, \psi\rangle \xrightarrow{\text{SELECT}} \langle fin[\![\text{SELECT}]\!], \psi'\rangle\}\\
&\text{where } \psi' = \bigwedge\left\{\overline{\mathsf{v}} \rightarrow \overline{\mathsf{v_a}.w_i} \mid \overline{\mathsf{v}} \in (\overline{\mathsf{V}}[\![\phi]\!] \cup \overline{\mathsf{V}}[\![\mathbf{exp}]\!] \cup \overline{\mathsf{V}}[\![\phi']\!] \cup \overline{\mathsf{V}}[\![\mathbf{exp'}]\!]) \wedge \overline{\mathsf{v_a}.w_i} \in \overline{\mathsf{v_a}.\boldsymbol{w}} \wedge \overline{\mathsf{v}} \neq \overline{\mathsf{v_a}.w_i}\right\}\wedge\\
&\quad \bigwedge\left\{\overline{\mathsf{v_i}} \rightarrow \overline{\mathsf{v_a}.w_i} \mid \overline{\mathsf{v_i}} \in \overline{\mathsf{V}}[\![u_i]\!] \wedge\ u_i \in \boldsymbol{u} \wedge \overline{\mathsf{v_a}.w_i} \in \overline{\mathsf{v_a}.\boldsymbol{w}}\right\} \wedge \left(\psi \ominus \bigwedge\{\overline{\mathsf{v}} \rightarrow \overline{\mathsf{v_a}.w_i} \mid \overline{\mathsf{v}} \in \overline{\mathsf{V}} \wedge \overline{\mathsf{v_a}.w_i} \in \overline{\mathsf{v_a}.\boldsymbol{w}}\}\right)\\
&\overline{\mathsf{T}}[\![\text{UPDATE}]\!]\\
&\stackrel{def}{=} \overline{\mathsf{T}}[\![\langle {}^{\ell'}\,\mathbf{v_d} \stackrel{upd}{=} \mathbf{exp},\ {}^{\ell}\phi\rangle]\!]\\
&\stackrel{def}{=} \{\langle \ell, \psi\rangle \xrightarrow{\text{UPDATE}} \langle fin[\![\text{UPDATE}]\!], \psi'\rangle\}\\
&\qquad\qquad\qquad\qquad \text{where } \psi' = \bigwedge\left\{\overline{\mathsf{v}}_1 \rightarrow \overline{\mathsf{v}}_2 \mid \overline{\mathsf{v}}_1 \in \overline{\mathsf{V}}[\![\phi]\!] \wedge \overline{\mathsf{v}}_2 \in \overline{\mathbf{v_d}}\right\}\wedge\\
&\qquad\qquad\qquad\qquad \bigwedge\left\{\overline{\mathsf{v_i}} \rightarrow \overline{\mathsf{v_j}} \mid \overline{\mathsf{v_i}} \in \overline{\mathsf{V}}[\![\mathsf{exp_i}]\!] \wedge\ \mathsf{exp_i} \in \mathbf{exp} \wedge \overline{\mathsf{v_j}} \in \overline{\mathbf{v_d}}\right\} \wedge \psi\\
&\overline{\mathsf{T}}[\![\text{INSERT}]\!]\\
&\stackrel{def}{=} \overline{\mathsf{T}}[\![\langle {}^{\ell'}\,\mathbf{v_d} \stackrel{new}{=} \mathbf{exp},\ {}^{\ell}\mathsf{true}\rangle]\!]\\
&\stackrel{def}{=} \{\langle \ell, \psi\rangle \xrightarrow{\text{INSERT}} \langle fin[\![\text{INSERT}]\!], \psi'\rangle\}\\
&\qquad\qquad\qquad\qquad \text{where } \psi' = \bigwedge\left\{\overline{\mathsf{v_i}} \rightarrow \overline{\mathsf{v_j}} \mid \overline{\mathsf{v_i}} \in \overline{\mathsf{V}}[\![\mathsf{exp_i}]\!] \wedge \mathsf{exp_i} \in \mathbf{exp} \wedge \overline{\mathsf{v_j}} \in \overline{\mathbf{v_d}}\right\} \wedge \psi\\
&\overline{\mathsf{T}}[\![\text{DELETE}]\!]\\
&\stackrel{def}{=} \overline{\mathsf{T}}[\![\langle {}^{\ell'} del(\mathbf{v_d}),\ {}^{\ell}\phi\rangle]\!]\\
&\stackrel{def}{=} \{\langle \ell, \psi\rangle \xrightarrow{\text{DELETE}} \langle fin[\![\text{DELETE}]\!], \psi'\rangle\}\\
&\qquad\qquad\qquad\qquad \text{where } \psi' = \bigwedge\left\{\overline{\mathsf{v}}_1 \rightarrow \overline{\mathsf{v}}_2 \mid \overline{\mathsf{v}}_1 \in \overline{\mathsf{V}}[\![\phi]\!] \wedge \overline{\mathsf{v}}_2 \in \overline{\mathbf{v_d}}\right\} \wedge \psi
\end{aligned}
$$

Table 17. Database dB

(a) Table "*Emp*"

ID	*Name*	*Sal*
1	Alberto	1110
2	Matteo	1638
3	Francesco	2255
4	Smith	1840

(b) Table "*Job*"

Type	*Rank*	*BASIC*	*HRA*	*DA*
Security	S1	800	20	65
Security	S2	600	20	65
Security	S3	520	15	65
Technical	T1	1000	25	75
Technical	T2	920	25	75
Technical	T3	880	20	75
Technical	T4	840	20	75
Admin	A1	1240	25	80
Admin	A2	1100	25	80

The following logical formulae representing PD-dependences exist in q_5:

$$\psi_5 = \overline{Sal} \rightarrow \overline{\mathsf{v_a}.w_1},\ \overline{BASIC} \rightarrow \overline{\mathsf{v_a}.w_1},\ \overline{DA} \rightarrow \overline{\mathsf{v_a}.w_1},\ \overline{HRA} \rightarrow \overline{\mathsf{v_a}.w_1},\ \overline{Type} \rightarrow \overline{\mathsf{v_a}.w_1}$$

Assuming 'Sal', '$BASIC$', 'HRA', 'DA' are private and at least one employee in each job-type must exist, we see that although syntactic PD-dependences above indicating the presence of information leakage, but in practice nothing about these secrets is leaked through $\mathsf{v_a}.w_1$.

Here is an another example of PD-dependence that is indicating false alarm on leakage: consider the code $\{\$x = 4 * \$w *\ log\ 2;$ `UPDATE` t `SET` $a = a + \$x;\}$. Assuming $\$x$ is private and $\$w$, a are public, we see that the dependence $\overline{\$x} \rightarrow \overline{a}$ generates false alarm as because $\$x$ is always equal to 0.

Table 18. Database $dB^\sharp$

(a) Table "*Emp*$^\sharp$"

ID$^\sharp$	*Name*$^\sharp$	*Sal*$^\sharp$
1	Alberto	[1110, 1110]
2	Matteo	[1638, 1638]
3	Francesco	[2255, 2255]
4	Smith	[1840, 1840]

(b) Table "*Job*$^\sharp$"

Type$^\sharp$	*Rank*$^\sharp$	*BASIC*$^\sharp$	*HRA*$^\sharp$	*DA*$^\sharp$
Security	[S1, S3]	[520, 800]	[15, 20]	[65, 65]
Technical	[T1, T4]	[840, 1000]	[20, 25]	[75, 75]
Admin	[A1, A2]	[1100, 1240]	[25, 25]	[80, 80]

To remove all such false alarms and to increase the accuracy of the analysis, we analyze programs by using the semantic-based abstract interpretation framework.

Consider an abstract domain $\aleph$ where numerical attributes and numerical application variables are abstracted by the *domain of intervals*[4]. The abstraction yields an abstract query ${\sf q_6}^\sharp$ corresponding to ${\sf q_6}$ and an abstract database depicted in Table 18.

$$\begin{aligned}{\sf q_6}^\sharp = \texttt{SELECT}^\sharp \quad Type^\sharp \quad \texttt{INTO}^\sharp \quad {\sf v_a}^\sharp \quad \texttt{FROM}^\sharp \quad Emp^\sharp,\\ Job^\sharp \quad \texttt{WHERE}^\sharp \quad Sal^\sharp \quad =^\sharp \quad BASIC^\sharp + (BASIC^\sharp \; * \\ (DA^\sharp/[100,100])) + (BASIC^\sharp * (HRA^\sharp/[100,100]))\end{aligned}$$

The right-hand side expression of the condition in $\texttt{WHERE}^\sharp$ is evaluated to abstract values $[936, 1480]$, $[1638, 2000]$, and $[2255, 2542]$ respectively corresponding to the three abstract tuples in "$Job^\sharp$". Observe that, according to the assumption that at least one employee must exist in each job-type, there exist at least one '$Sal^\sharp$' in "$Emp^\sharp$" for which "$Sal^\sharp =^\sharp [936, 1480]$" is true, according to the following:

$$[l_i, h_i] =^\sharp [l_j, h_j] \triangleq \begin{cases} \text{true} & \text{if } (l_i \geq l_j \wedge h_i \leq h_j) \\ \text{false} & \text{if } h_i < l_j \vee l_i > h_j \\ \top & \text{otherwise} \end{cases}$$

Similar for "$Sal^\sharp =^\sharp [1638, 2000]$" and "$Sal^\sharp =^\sharp [2255, 2542]$". Therefore, the evaluation of ${\sf q_6}$ on dB always gives the same result *w.r.t.* the property $\aleph$, irrespective of the states of "*Emp*".

We can perform similar analysis of the code $\{\$x = 4 * \$w * \; log\ 2;\ \texttt{UPDATE}\ t\ \texttt{SET}\ a = a + \$x;\}$ in the *domain of intervals*, yielding to "*no update*" of the values in public attribute a.

The interaction of the logical and numerical domains can be formalized by using the reduced Product $\mathsf{D}^\natural$ as follows:

$$\mathsf{D}^\natural = \{\varrho(\langle \mathsf{X}, \mathfrak{N}\rangle) \mid \mathsf{X} \in \wp(\Sigma^{\star\sharp}), \mathfrak{N} \in \aleph\}$$

where $\varrho(\langle \mathsf{X}, \mathfrak{N}\rangle) = \big\{\langle \ell_i, \psi_k\rangle \mid \langle \ell_i, \psi_j\rangle \in \mathsf{X} \wedge \psi_k = (\psi_j \ominus \{\bar{\mathsf{v}}_1 \rightarrow \bar{\mathsf{v}}_2 \mid y \in \gamma(\mathfrak{N})\})\big\}$.

In the example above, by analyzing ${\sf q_6}$ in the abstract domain $\aleph$ where numerical variables are abstracted by the *domain of intervals*, we see that the value of ${\sf v_a}.w_1$ generated by ${\sf q_6}$ is always constant throughout the program execution

[4] For other type of variables, the abstraction function represents identity function.

w.r.t. $\aleph$. As $\psi_6 \in \texttt{Pos}$ and $\mathsf{v_a}.w_1 \in \aleph$, the reduced product operator ϱ removes from ψ_6 all dependences in the form "$x \rightarrow \mathsf{v_a}.w_1$" (that are representing false alarms), and makes the analysis more accurate and efficient.

5 Implementing the Analysis in Sails

In this section, we present Sails. The tool is an instance of the generic analyzer Sample. This is why we discuss the main issues we have to solve in order to deal with information leakage analysis within Sample.

5.1 Sample

Sample (Static Analyzer of Multiple Programming LanguagEs) is a generic analyzer based on the abstract interpretation theory. Relying on compositional analyses, Sample can be plugged with different heap abstractions, approximations of other semantic information (e.g., numeric domains or information flow), properties of interest, and languages. Several heap analyses, semantic and numerical domains have been already plugged. The analyzer works on an intermediate language called Simple. Up to now, Sample supports the compilation of Scala and Java bytecode to Simple.

Figure 3 depicts the overall structure of Sample. Source code programs are compiled to Simple. A fixpoint engine receives a heap analysis, a semantic domain, and a control flow graph (whose blocks are composed by a sequence of Simple statements), and it produces an abstract result over the control flow graph of each method. This result is passed to a property checker that produces some output (e.g., warnings) to the user. The integration of an analysis in Sample allows one to take advantage of all aspects not strictly related to the analysis but that can improve its final precision (e.g., heap or numerical abstractions). For instance, Sample is interfaced with the Apron library [28] and contains a heap analysis based of TVLA [39].

5.2 Heap Abstraction

In Sample heap locations are approximated by abstract heap identifiers. While the identifiers of program variables are fixed and represent exactly one concrete variable, the abstract heap identifiers may represent several concrete heap locations (e.g., if they summarize a potentially unbounded list), and they can be merged and split during the analysis. In particular we have to support (i) assignments on summary heap identifiers, and (ii) renaming of identifiers.

In order to preserve the soundness of Sails, we have to perform weak assignments on summary heap identifiers. Since a summary abstract identifier may represent several concrete heap locations and only one of them would be assigned in one particular execution, we have to take the upper bound between the assigned value, and the old one.

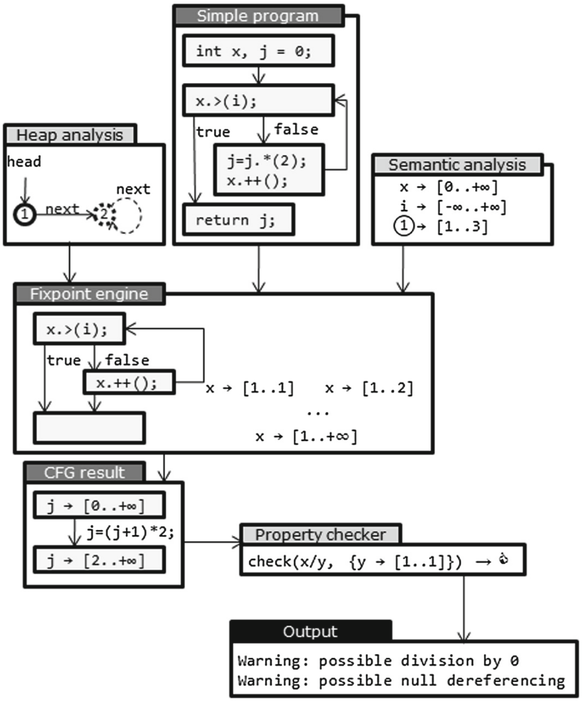

Fig. 3. The structure of Sample

The heap abstraction could require to rename, summarize or split existing identifiers. This information is passed through a replacement function $rep : \wp(\mathsf{Id}) \to \wp(\mathsf{Id})$, where Id is the set containing all heap identifiers. For instance, in TVLA two abstract nodes represented by identifiers a_1 and a_2 may be merged to a summary node a_3, or a summary abstract node b_1 may be splitted to b_2 and b_3. Our heap analysis will pass $\{a_1, a_2\} \mapsto \{a_3\}$ and $\{b_1\} \mapsto \{b_2, b_3\}$ to Sails in these cases, respectively. Given a single replacement $\mathsf{S}_1 \mapsto \mathsf{S}_2$, Sails removes all subformulae dealing with some of the variables in S_1, and for each removed subformula s it inserts a new subformula s' renaming each of the variables in S_1 to each of the variables in S_2. Formally:

$$
\begin{aligned}
&rename : (\mathsf{Pos} \times (\wp(\mathsf{Id}) \to \wp(\mathsf{Id}))) \to \mathsf{Pos}\\
&rename(\sigma^\sharp, rep) = \{(\mathsf{i'}_1, \mathsf{i'}_2) : (\mathsf{i}_1, \mathsf{i}_2) \in \sigma^\sharp \wedge\\
&\quad \mathsf{i'}_1 = \begin{cases} \mathsf{i}_1 & \text{if } \nexists \mathsf{R}_1 \in dom(rep) : \mathsf{i}_1 \in \mathsf{R}_1\\ \mathsf{k}_1 & \text{if } \exists \mathsf{R}_1 \in dom(rep) : \mathsf{i}_1 \in \mathsf{R}_1 \wedge \mathsf{k}_1 \in rep(\mathsf{R}_1)\end{cases},\\
&\quad \mathsf{i'}_2 = \begin{cases} \mathsf{i}_2 & \text{if } \nexists \mathsf{R}_2 \in dom(rep) : \mathsf{i}_2 \in \mathsf{R}_2\\ \mathsf{k}_2 & \text{if } \exists \mathsf{R}_2 \in dom(rep) : \mathsf{i}_2 \in \mathsf{R}_2 \wedge \mathsf{k}_2 \in rep(\mathsf{R}_2)\end{cases} \}
\end{aligned}
$$

5.3 Propositional Formulae

We have to introduce some slight modifications on the domain for information leakage analysis described in Sect. 2 to work with object oriented languages. We can consider a propositional formula ϕ as a conjunction of subformulae $(\zeta_0 \wedge \ldots \wedge \zeta_n)$. In the implementation, each subformula is an implication between two identifiers. Then we represent a subformula as a pair of identifiers and a formula as a set of subformulae. Consider the statement $\mathsf{if}(\mathsf{x} > 0)\ \mathsf{y} = \mathsf{z};$. The formula obtained after the analysis of this statement is represented by the set $\{(\bar{\mathsf{y}}, \bar{\mathsf{z}}), (\bar{\mathsf{x}}, \bar{\mathsf{y}})\}$, where we denote the identifier of the variable u by $\bar{\mathsf{u}}$. The order relation "$\trianglelefteq$" is defined by the subset relation $(\phi_0 \trianglelefteq \phi_1 \Leftrightarrow \phi_0 \subseteq \phi_1)$.
Consequently, in the implementation the set of propositional variables $\overline{\mathsf{V}}$ consists in the set of identifier Id, a single propositional formula is represented by $\wp(\mathsf{Id} \times \mathsf{Id})$ and an abstract state $\sigma^\sharp \in \Sigma^\sharp$ is a conjunction of propositional formulae represented by $\wp(\wp(\mathsf{Id} \times \mathsf{Id}))$.

5.4 Implicit Flow Detection

An implicit information flow occurs when there is an information leakage from a variable in a condition to a variable assigned inside a block dependent on that condition. For instance, in $\mathsf{if}(\mathsf{x} > 0)\ \mathsf{y} = \mathsf{z};$ there is an explicit flow from z to y, and an implicit flow from x to y. To record these relations we relate the variables in the conditions to the variables that have been assigned in the block. When we join two blocks coming from the same condition, we discharge all implicit flows on the abstract state. Observe that Sails does not support all cfgs that can be represented in Sample but only the ones coming from structured programs, i.e., that corresponds to programs with if and while statements and not with arbitrary jumps like goto.

5.5 Property

An information flow analysis can be carried out by considering different attacker abilities. We implemented two scenarios: when the attacker can read public variables only at the beginning and at the end of the computation, and when the attacker can read public variables after each step of the computation[5]. Moreover, we implemented two security properties for each attacker: secrecy (i.e., information leakage analysis) and integrity.

The verification of these properties happens after the computation of the analysis and the declaration of private variables (at run time, by a text files writing the variables name or by a graphical user interface selecting the variables in a list).

[5] Notice that, as in [47], we assume that the attacker, in both cases, knows the source code of the program.

5.6 Numerical Analysis

The information flow analysis is based on the reduced product of a dependency and a numerical analysis. Thanks to the compositional structure of Sample, we can plug Sails with different numerical domains. In particular, Sample supports the Apron library. In this way, we can combine Sails with all numerical domains contained in Apron (namely, Polka, the Parma Polyhedra Library, Octagons, and a deep implementation of Intervals).

In addition, we can apply different heap abstractions. For instance, if we are not interested to the heap structure, we can use a less accurate domain that approximates all heap locations with one unique summary node, as we will do in Sect. 6.2.

5.7 Complexity of the Analysis

The complexity of variables dependency analysis showed in Sect. 2 is strictly correlated to the complexity of propositional formulae. Logical domains, in literature, are widely treated and generally, the logical equivalence of two boolean expression is a co-NP-complete problem. However, this complexity issue may not matter much in practice because the size of the set of variables appearing in the program is reasonably small. Hence, on the one hand, work with propositional formulae requires the solving of a co-NP-complete problem, while on the other hand, in many frameworks (included our system), Pos only deal with the variables appearing in the programs, reducing in this way the complexity. Generally, it is possible to increase the efficiency of the computation using the *binary decision diagrams* (BDDs) for the implementation of propositional formulae. For more information about binary decision diagrams see [1].

The simplification adopted in the implementation, i.e. the definition of "$\trianglelefteq$" by the subset relation ($\phi_0 \trianglelefteq \phi_1 \Leftrightarrow \phi_0 \subseteq \phi_1$), permits to decrease the complexity. In fact, decreasing the precision of the analysis, we can compare two propositional formulae in polynomial time.

About polyhedra analysis, the complexity is well and completely treated in many works [5] and heavily depends on its implementation. For example many implementations, e.g. Polylib and New Polka, use matrices of coefficients, that cannot grow dynamically, and the worst case space complexity of the methods employed is exponential. In PPL library, instead, all data structures are fully dynamic and automatically expanded (in amortized constant time) ensuring the best use of available memory. Comparing the efficiency of polyhedra libraries is not a simple task, because the pay-off depends on the targeted applications: in [5] the authors presented many test results about it.

The complexity of reduced product, and more precisely of reduction operator presented in Sect. 3.1, is strictly connected with the complexity of the operations on the domains we combine.

6 Experimental Results

In this section, we present the experimental results of Sails. First of all, we present the results in terms of precision when we analyze a case study involving recursive data structures. Then, we present the results obtained when applying Sails to the SecuriBench-micro suite.

6.1 Case Study

Consider the Java code in Fig. 4. Class ListWorkers models a list of workers of an enterprise. Each node contains the salary earned by the worker, and some other data (e.g., name and surname of the person). Method updateSalaries is defined as well. It receives a list of employees and a list of managers. These two lists are supposed to be disjoint. First method updateSalaries computes the maximal salary of an employee. Then it traverses the list of managers updating their salary to the maximal salary of employees if manager's salary is less than that.

Usually managers would not like to leak information about their salary to employees (secrecy property). This property could be expressed in Sails specifying that we do not want to have a flow of information *from* managers *to* employees. More precisely, we want to prove the absence of information leakage from the content of field salary of any node reachable from managers to any node reachable from employees.

```
class ListWorkers {
  int salary;
  ListWorkers next;
  ...
}

public void updateSalaries
(ListWorkers employees, ListWorkers managers){
  int maxSalary = 0;
  ListWorkers it=employees;
  while(it!=null) {
    if(it.salary>maxSalary)
      maxSalary=it.salary;
    it=it.next;
  }
  it=managers;
  while(it!=null) {
    if(it.salary < maxSalary)
      it.salary=maxSalary;
    it=it.next;
  }
}
```

Fig. 4. A motivating example

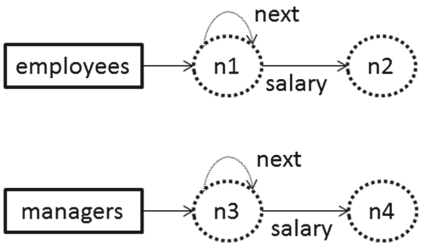

Fig. 5. The initial state of the heap abstraction

We combine Sails with a heap analysis that approximates all objects created by a program point with a single abstract node [20]. We start the analysis of method updateSalaries with an abstract heap in which lists managers and employees are abstracted with a summary node and they are disjoint. Figure 5 depicts the initial state, where n2 and n4 contains the salary values of the ListWorkers n1 and n3, respectively. In the graphic representation we adopt dotted circles to represent summary nodes, rectangles to represent local variables, and edges between nodes to represent what is pointed by local variables or fields of objects. Note that the structure of these two lists does not change during the analysis of the program, since method updateSalaries does not modify the heap structure.

Sails infers that, after the first while loop at line 15, there is a flow of information from n2 to maxSalary. This happens because variable it points to n1 before the loop (because of the assignment at line 9), and it iterates following field next (obtaining always the summary node n1) perhaps assigning the content of it.salary (that is, node n2) to maxSalary. Therefore, at line 15 we have the propositional formula n2 $\rightarrow$ maxSalary.

Then updateSalaries traverses the managers list. For each node, it could assign maxSalary to it.salary. Similarly to what happened in the previous loop, variable it points to n3 before and inside the loop, since field next always points to the summary node n3. Therefore the assignment at line 18 could potentially affects only node n4. For this reason, Sails discovers a flow of information from maxSalary to n4, represented by the propositional formula maxSalary $\rightarrow$ n4.

At the end of the analysis, Sails soundly computes that (n2 $\rightarrow$ maxSalary) $\wedge$ (maxSalary $\rightarrow$ n4). By the transitive property, we know that there could be a flow of information from n2 to n4, that is, from employees to managers. This flow is allowed by our security policy. On the other hand, we also discovered that there is no information leakage from list managers to list employees, since Sails does not contain any propositional formula with this flow. Therefore Sails proves that this program is safe.

"Noninterference of programs essentially means that a *variable* of confidential (high) input does not cause a variation of public (low) output" [38]. Thanks to the combination between a heap abstraction and an abstract domain tracking information flow, Sails deals directly with the structure of the heap, extending the concept of noninterference from variables to portions of the heap represented by abstract nodes. This opens a new scenario since we can prove that a whole data structure does not interfere with another one, as we have done in this example. As far as we know, Sails is the only tool that performs a noninterference analysis over a heap abstraction, and therefore it can prove properties like "there is no information flow from the nodes reachable from v_1 to the nodes reachable from v_2".

6.2 Benchmarks

A well-established way of studying the precision and the efficiency of information flow analyses is the SecuriBench-micro suite [45]. We applied Sails to this test suite; the description and the results of these benchmarks are reported in Table 19. Column `fa` reports if the analysis did not produce any false alarm. We combined Sails with a really rough heap abstraction that approximates all

Table 19. SecuriBench-micro suite

Name	Description	fa
`Aliasing1`	Simple aliasing	✓
`Aliasing2`	Aliasing false positive	✓
`Basic1`	Very simple XSS	✓
`Basic2`	XSS combined with a conditional	✓
`Basic3`	Simple derived integer test	✓
`Basic5`	Test of derived integer	✓
`Basic6`	Complex test of derived integer	✓
`Basic8`	Test of complex conditionals	✓
`Basic9`	Chains of value assignments	✓
`Basic10`	Chains of value assignments	✓
`Basic11`	A simple false positive	✓
`Basic12`	A simple conditional	✓
`Basic18`	Protect agains simple loop unrolling	✓
`Basic28`	Complicated control flow	✓
`Pred1`	Simple if(false) test	✗
`Pred2`	Simple correlated tests	✓
`Pred3`	Simple correlated tests	✓
`Pred4`	Test with an integer variable	✓
`Pred5`	Test with a complex conditional	✓
`Pred6`	Test with addition	✗
`Pred7`	Test with multiple variables	✗

Table 20. Jif case studies

Name	Description	fa
A	Simple explicit flow test	✓
Account	Simple explicit flow test	✓
ConditionalLeak	Explicit flow in if statement	✓
Do	Implicit flow in the loop	✓
Do2	Implicit flow if and loop	✓
Do3	Implicit flow loop and if	✓
Do4	Implicit flow loop and if	✓
Do5	Implicit flow loop and if	✓
If1	Simple implicit flow	✓
Implicit	Simple implicit flow	✓

concrete heap locations with one abstract node. Sails detected all information leakages in all tests, but in three cases (`Pred1`, `Pred6` and `Pred7`) it produced false alarms. This happens because Sails abstracts away the information produced when testing to true or false boolean conditions in if or while statements.

Since these benchmarks cover only problems with explicit flows, we performed further experiments using some Jif [36] case studies. The results are reported in Table 20: we discovered all flows without producing any false alarm.

These results allow us to conclude that Sails is precise, since in 90% of the cases (28 out of 31 programs) it does not produce any false alarm.

About the performances, the analysis of all case studies takes 1.092 s (0.035 s per method in average) without combining it with a numerical domain. When we combine it with Intervals it takes 3.015 s, whereas it takes 6.130 s in combination with Polka. All tests are performed using a MacBook Pro Intel Core 2 Duo 2.53 GHz with 4 GB of RAM memory. Therefore the experimental results underline the efficiency of Sails as well.

7 Related Work

In a security-typed language Volpano et al. [46] were the first ones to develop a type system to enforce information flow policies, where a type is inductively associated at compile-time with program statements in such a way that well-typed programs satisfy the non-interference property. The authors formulated the certification conditions of Denning's analysis [18] as a simple type system for a deterministic language: basically, a formal system of type inference rules for making judgments about programs. More generally, type-based approaches are designed such that well-typed programs do not leak secrets. A type is inductively associated at compile-time with program statements in such a way that any statement showing a potential low disclosing secrets is rejected. Type systems that enforce secure information flow have been designed for various languages and they have been used in different applications. Some of these approaches are, for example, applied to specific programs, e.g., written in VHDL [44], where the analysis of information flow is closely related to the context. Moreover, the secure information flow problem was also handled in different situation, for example with multi-threaded programs [42] or with programs that employ explicit cryptographic operations [3,21].

A different approach is the use of standard control flow analysis to detect information leakage, e.g., [9,29,30]. The idea, of this technique, is to conservatively find the program paths through which data may flow. Generally, the data flow analysis approach to secure information flow as a translation from a given program that captures and facilitates reasoning about the possible flows. For example, Leino and Joshi [29] showed an application based on semantics, deriving a first-order predicate whose validity implies that an attacker cannot deduce any secure information from observing the public inputs, outputs and termination behavior of the program.

The use of abstract interpretation in language-based security is not new, even though there aren't many works that use the lattice of abstract interpretations for evaluating the security of programs (for example [49]).

Probably, the main work about information flow analysis by abstract interpretation was done by Giacobazzi and Mastroeni [22] that generalizes the notion of non-interference making it parametric relatively to what an attacker can observe, and using it to model attackers as abstractions. A program semantics was characterized as an abstract interpretation of its maximal trace semantics in the corresponding transition system. The authors gave a method for checking abstract non-interference and they proved that checking abstract non-interference is a standard static program analysis problem. This method allows both to compare attackers and program secrecy by comparing the corresponding abstractions in the lattice of abstract interpretations, and to design automatic program certification tools for language-based security.

There are not so many implementations of secure information flow. In early 2000, some works began the control of sensitive information in realistic languages [7,37]. Jif [4] and Flow CAML [40] are, as far as we know, the two main implementations about information flow analysis. Notice that, in the last years other language-based tools are developed for some specific language, e.g., Fabric [32] for distributed computing, the LIO library in haskell [43] and FlowFox [16] a tool for JavaScript.

According to [41], it seems be helpful to distinguish between two different application scenarios: *developing secure software* and *stopping malicious software*. The first scenario is based on to secure information flow analysis to help the development of software that satisfies some security properties. In this case, the analysis serves as a program development tool. The static analysis tool would alert the programmer to potential leaks and the developer could rewriting the code as necessary. An example of this scenario can be found in [4], where Askarov and Sabelfeld discusses the implementation of a "mental poker" protocol in Jif. The second scenario, instead, the secure information flow analysis is used as a kind of filter to stop malicious software. In this case, we might imagine analyzing a piece of untrusted code before executing it, with the goal of guaranteeing its safety. This is much more challenging than first scenario: probably we would not have access to the source code and we would need to analyze the binary code. Analyzing binaries is more difficult than analyzing source code and has not received much attention in the literature (a Java bytecodes analysis is performed, for instance, by Barthe and Rezk in [8]).

Given this overall context, the approach adopted in Sails is quite different from existing tools that deal with information flow analysis. Jif, for example, is a security-typed programming language that extends Java with support for information flow and access control, enforced at compile time and it is an ad hoc analysis that requires to annotate the code with some type information. If on the one hand Jif is more efficient than Sails, on the other hand Sails does not require any manual annotation, and it takes all advantages of compositional analyzers (e.g., we can combine Sails with a TVLA-based heap abstraction).

Our approach does not require to change the programming language, since it infers the flow of information directly on the original program, and it asks what are the private data that have not to be leaked to the user during the analysis execution.

8 Conclusions

In this paper we presented an information flow analysis through abstract interpretation based on a new domain that combines a variable dependency analysis and a numerical domain. We then introduced Sails that applies and implements this analysis on object-oriented programs. Sails is an extension of Sample, therefore it is modular with respect to the heap abstraction, and it can verify noninterference over recursive data structures using simple and efficient heap analyses. The experimental results underline the effectiveness of the analysis, since Sails is in position to analyze several benchmarks in few milliseconds per program without producing false alarms in more than 90% of the programs. Moreover, our tool does not require to modify the original language, since it works with mainstream languages like Java, and it does not require any manual annotation.

Acknowledgments. This work has been partially supported by CINI Cybersecurity National Laboratory within the project "FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks" funded by CISCO Systems Inc. and Leonardo SpA, and by MIUR-MAE within the Project "Formal Specification for Secured Software System", under the Indo-Italian Executive Programme of Cooperation in Scientific and Technological Cooperation Project number IN17MO07.

References

1. Andersen, H.R.: An introduction to binary decision diagrams. Technical report, Course Notes on the WWW (1997)
2. Armstrong, T., Marriott, K., Schachte, P., Søndergaard, H.: Two classes of boolean functions for dependency analysis. Sci. Comput. Program. **31**, 3–45 (1998)
3. Askarov, A., Hedin, D., Sabelfeld, A.: Cryptographically-masked flows. Theor. Comput. Sci. **402**, 82–101 (2008)
4. Askarov, A., Sabelfeld, A.: Security-typed languages for implementation of cryptographic protocols: a case study. In: di Vimercati, S.C., Syverson, P., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 197–221. Springer, Heidelberg (2005). https://doi.org/10.1007/11555827_12
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**, 3–21 (2008)
6. Bagnara, R., Hill, P.M., Zaffanella, E.: Applications of polyhedral computations to the analysis and verification of hardware and software systems. Theor. Comput. Sci. **410**, 4672–4691 (2009)
7. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a Java-like language. In: Proceedings of the 15th IEEE Workshop on Computer Security Foundations, CSFW 2002. IEEE Computer Society, Washington, DC (2002)
8. Barthe, G., Rezk, T.: Non-interference for a JVM-like language. In: Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI 2005, pp. 103–112. ACM, New York (2005)
9. Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for secrecy and non-interference in networks of processes. In: Malyshkin, V. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 27–41. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44743-1_3
10. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 196–206. ACM, New York (2007)
11. Cortesi, A., Filé, G., Winsborough, W.H.: Prop revisited: propositional formula as abstract domain for groundness analysis. In: LICS, pp. 322–327 (1991)
12. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 505–521. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24559-6_34
13. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS
AQ2 Press, Amsterdam (1999)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979, pp. 269–282. ACM, New York (1979)
15. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1978, pp. 84–96. ACM, New York (1978)
16. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a web browser with flexible and precise information flow control. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012). ACM (2012)

17. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**, 236–243 (1976)
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**, 504–513 (1977)
19. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE-2010. LNCS, vol. 6117, pp. 186–200. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13464-7_15
20. Ferrara, P.: A fast and precise alias analysis for data race detection. In: Proceedings of the Third Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2008), Electronic Notes in Theoretical Computer Science. Elsevier, April 2008
21. Focardi, R., Centenaro, M.: Information flow security of multi-threaded distributed programs. In: Proceedings of the third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2008, pp. 113–124. ACM, New York (2008)
22. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 186–197. ACM, New York (2004)
23. Halder, R., Cortesi, A.: Abstract interpretation of database query languages. Comput. Lang. Syst. Struct. **38**, 123–157 (2012)
24. Halder, R., Cortesi, A.: Abstract program slicing of database query languages. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, pp. 838–845. ACM Press (2013)
25. Halder, R., Zanioli, M., Cortesi, A.: Information leakage analysis of database query languages. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, Gyeongju, Korea, pp. 813–820. ACM Press, 24–28 March 2014
26. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. Wiley, New York (1990)
27. Jeannet, B.: Convex polyhedra library, March 2002. Documentation of the "New Polka" library. http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html
28. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
29. Joshi, R., Rustan, K., Leino, M.: A semantic approach to secure information flow. Sci. Comput. Program. **37**, 113–138 (2000)
30. Laud, P.: Semantics and program analysis of computationally secure information flow. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 77–91. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45309-1_6
31. Li, P., Zdancewic, S.: Arrows for secure information flow. Theor. Comput. Sci. **411**, 1974–1994 (2010)
32. Liu, J.D., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: a platform for secure distributed computation and storage. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 321–334. ACM, New York (2009)
33. Liu, Y., Milanova, A.: Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering, CSMR 2010, pp. 146–155. IEEE Computer Society, Washington, DC (2010)

34. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44978-7_10
35. Miné, A.: The octagon abstract domain. In: Proceedings of the Workshop on Analysis, Slicing, and Transformation (AST 2001), pp. 310–319. IEEE CS Press, October 2001
36. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: JIF: Java information flow. Software release, July 2001–2004
37. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. **25**, 117–158 (2003)
38. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
39. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**, 217–298 (2002)
40. Simonet, V.: The flow Caml System: documentation and user's manual. Technical report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003
41. Smith, G.: Principles of secure information flow analysis. In: Malware Detection, pp. 297–307 (2007)
42. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, pp. 355–364. ACM, New York (1998)
43. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. SIGPLAN Not. **46**(12), 95–106 (2011)
44. Tolstrup, T.K., Nielson, F., Nielson, H.R.: Information flow analysis for VHDL. In: Malyshkin, V. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 79–98. Springer, Heidelberg (2005). https://doi.org/10.1007/11535294_8
45. Stanford University. Stanford SecuriBench Micro. http://suif.stanford.edu/~livshits/work/securibench-micro/
46. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**, 167–187 (1996)
47. Zanioli, M., Cortesi, A.: Information leakage analysis by abstract interpretation. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Královič, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 545–557. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18381-2_45
48. Zanioli, M., Ferrara, P., Cortesi, A.: Sails: static analysis of information leakage with sample. In: Proceedings of the 2012 ACM Symposium on Applied Computing, pp. 1308–1313. ACM Press (2012)
49. Zanotti, M.: Security typings by abstract interpretation. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 360–375. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45789-5_26

Author Queries

Chapter 6

Query Refs.	Details Required	Author's response
AQ1	Please confirm if the corresponding author and mail id are correctly identified. Amend if necessary.	
AQ2	References [13, 31] are given in the list but not cited in the text. Please cite them in text or delete them from the list.	