

Information Leakage Analysis by Abstract Interpretation

Matteo Zanioli^{1,2} and Agostino Cortesi¹

¹ Università Ca' Foscari Venezia

² Université Paris Diderot (Paris 7)

Abstract. Protecting the confidentiality of information stored in a computer system or transmitted over a public network is a relevant problem in computer security. The approach of information flow analysis involves performing a static analysis of the program with the aim of proving that there will not be leaks of sensitive information. In this paper we propose a new domain that combines variable dependency analysis, based on propositional formulas, and variables' value analysis, based on polyhedra. The resulting analysis is strictly more accurate than the state of the art abstract interpretation based analyses for information leakage detection. Its modular construction allows to deal with the tradeoff between efficiency and accuracy by tuning the granularity of the abstraction and the complexity of the abstract operators.

1 Introduction

Protecting the confidentiality of information stored in a computer system or transmitted over a public network is a relevant problem in computer security.

Any information flow analysis involves performing a static analysis of the program with the aim of proving that there will not be leaks of sensitive information. There is an information flow from object x to object y whenever the information stored in x is transferred to, or used to derive information transferred to, object y . Flows would be explicit or implicit. An explicit flow occurs whenever the operations generating it are independent of the value of x ; whereas an implicit flow occurs whenever a statement specifies a flow from some arbitrary z to y , but the execution depends on the value of x . The starting point in secure information flow analysis is the classification of program variables into different security levels. In the simplest case, two levels are used: public (or low, L) and secret (or high, H). The main purpose is to prevent leak of sensitive information from an high variable to a lower one. More generally, we might work with a lattice of security levels, and we would aim to ensure that sensitive information flows only upwards in the lattice [8].

In 1982 Goguen and Meseguer introduced in [10] the notion of non-interference: *“one group of users/processes/variables, using a certain set of commands, is non-interfering with another group of users if what the first group does with those commands has no effect on what the second group of users/processes/variables*

can see". The idea behind non-interference is that someone observing the final values of public variables cannot conclude anything about the initial values of secret variables [14].

There is a widespread literature on methods and techniques for checking secure information flows in software: from standard control flow analysis to type inference. In a security-typed language Volpano, Irvine and Smith [16] were the first 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. Moreover, the same problem was handled also in different situation, for example with multi-threaded programs or with programs that employ explicit cryptographic operations.

A different approach is the use of standard control flow analysis to detect information leakage, e.g. [2]. Some of these works are applied to specific system, e.g. mobile ambients [3], or to specific programs, e.g. written in VHDL [15], where the analysis of information flow is closely related to the context.

The use of abstract interpretation in language-based security is not new, even though there aren't many work that use the lattice of abstract interpretations for evaluating the security of programs. Giacobazzi and Mastroeni in [9] generalize the notion of non-interference making it parametric relatively to what an attacker can observe and use it to model attackers as abstract interpretations.

In this paper we present an information flow analysis by abstract interpretation through a combination of two analysis: a syntactic variable dependency analysis, based on propositional formulas domain, and a variable value dependency using a Polyhedra analysis. An interesting aspect is that the polyhedra analysis can be replaced with other kinds of analysis which use different domains to represent the relations among variables' values.

The idea is to use logic formulas to represent dependency between variables, refine the analysis in order to reduce as much as possible "false alarms" and detect information leakages evaluating formulas on truth-assignment functions.

The analysis of a program involves the following steps:

- For each program instruction construct: a propositional formula (ϕ_i), through a fixpoint algorithm, which show an over-approximation of dependencies that occur between variables, and a polyhedron (\mathcal{P}_i) which represent an over-approximation of dependencies among variables value, through a classical polyhedra analysis.
- Refine each propositional formulas ϕ_i through the information in \mathcal{P}_i .
- Consider the public/private partitions of variables and the truth-assignment function \overline{T} , that assigns to a propositional variable the value T (true) or the value F (false) if the corresponding variable is respectively private or public. If \overline{T} does not satisfy ϕ_i , there could be some information leakages.

In order to better understand how our new dependency analysis works, consider the following example.

Example 1. When a credit card PIN reaches the issuing bank, its correspondence with the validation data (i.e. the user PAN, and possibly other public data, such

as the card expiration date or the customer name) is checked via a verification API. Consider the case study showed in [4], where a strict subset of the real PIN verification function named `Encrypted_PIN_Verify` is considered.

This function checks the equality of the actual user PIN and the trial PIN inserted at the ATM, and it returns either the result of the verification or an error code. The former PIN is derived through the PIN derivation key pdk and from the public data $offset$, $vdata$, $dectab$, while the latter comes encrypted under the key k as EPB (Encrypted PIN Block). Variable $counter$ counts the number of executed test. Note that the two keys are pre-loaded in the HSM (Hardware Security Module), and they are never exposed to the untrusted external environment.

```

PIN_V( $PAN, EPB, len, offset, vdata, dectab, counter$ ) {
   $x_1 := enc_{pdk}(vdata)$ ;
   $x_2 := left(len, x_1)$ ;
   $x_3 := decimalize(dectab, x_2)$ ;
   $x_4 := sum\_mod10(x_3, offset)$ ;
   $x_5 := dec_k(EPB)$ ;
   $x_6 := fcheck(x_5)$ ;
  if ( $x_4 = x_6$ ) {
     $counter := counter + 1$ ;  $result := "PIN correct"$ ;
  } else {
     $counter := counter + 1$ ;  $result := "PIN wrong"$ ;
  }
  return  $result$ ;
}

```

When we apply our analysis we obtain, at the end of the program, the following formula, where the implication symbol can be read as possible information flow dependency.

$$\begin{aligned}
\phi = & (\overline{vdata} \rightarrow \overline{x_1}) \wedge (\overline{len} \rightarrow \overline{x_2}) \wedge (\overline{x_1} \rightarrow \overline{x_2}) \wedge (\overline{dectab} \rightarrow \overline{x_3}) \wedge (\overline{x_2} \rightarrow \overline{x_3}) \wedge \\
& (\overline{offset} \rightarrow \overline{x_4}) \wedge (\overline{x_3} \rightarrow \overline{x_4}) \wedge (\overline{EPB} \rightarrow \overline{x_5}) \wedge (\overline{x_5} \rightarrow \overline{x_6}) \wedge \\
& (\overline{x_6} \rightarrow \overline{result}) \wedge (\overline{x_4} \rightarrow \overline{result}) \wedge (\overline{x_6} \rightarrow \overline{result})
\end{aligned}$$

Let $\gamma : V \rightarrow \{L, H\}$ be a function that assign ‘‘L’’ class to $counter$ variable and ‘‘H’’ class to other variables. Through a traditional information flow analysis we would find as a false positive a warning of information leakage from variables x_4 and x_6 to variable $counter$, whereas with our analysis we obtain that $\overline{\gamma}$, the correspondent truth-assignment function, satisfies ϕ . In fact there is no information leakage in the program: the final value of variable $counter$ is independent from the value of variables x_4 and x_6 .

The rest of this paper is organized as follows. In the next two sections (Section 2 and 3) we define the concrete and the abstract domain, respectively, and in Section 4 we introduce our information leakage analysis. Finally, Section 5 concludes.

2 Concrete Domain

2.1 Syntax

We consider a simple imperative language where programs are labelled commands with the following syntax:

$$C ::= {}^\ell \text{skip} \mid {}^\ell x := E \mid C_1; C_2 \mid \text{if } {}^\ell B \text{ then } C_1 \text{ else } C_2 \text{ }^{\ell'} \text{endif} \mid \text{while } {}^\ell B \text{ do } C \text{ }^{\ell'} \text{done}$$

With E denoting expressions evaluated in the set of values \mathcal{V} with standard operations, i.e. if $\mathcal{V} = \mathbb{N}$ then E can be any arithmetical expression. In the following we will denote by $\mathbb{V}[[P]]$ the set of variables of the program P , by $i[[C]]$ and $f[[C]]$ the initial and final label of a command C respectively and by $\mathbb{A} = \{{}^\ell \text{skip}, {}^\ell x := E, {}^\ell B, {}^\ell \text{not} B, {}^\ell \text{endif}, {}^\ell \text{done}\}$, the set of actions.

2.2 Semantics

An environment $\rho \in \mathcal{E}$ is a function with signature: $\mathbb{V} \rightarrow \mathcal{V}$. A state $\sigma \in \Sigma \equiv \mathbb{L} \times \mathcal{E}$ is a pair $\langle \ell, \rho \rangle$ where the environment $\rho \in \mathcal{E}$ defines the current value $\rho(x)$ of the program variable $x \in \mathbb{V}[[C]]$ and the program label $\ell \in \mathbb{L}$ specifies which part of the program remains to be executed. We denote by $\mathbb{E}[[E]]\rho$ and $\mathbb{B}[[B]]\rho$ the expression and the condition evaluation of $E \in \mathbb{E}$ and $B \in \mathbb{B}$, respectively.

The execution of a program starts at its initial label with any possible value of the variables. Therefore, the set \mathcal{I} of possible initial states of a program P is $\mathcal{I}[[P]] \equiv \{\langle i[[P]], \rho \rangle \mid \rho \in \mathcal{E}\}$. In the same way, we can define $\mathcal{F}[[P]]$ as the set of possible final state of P : $\mathcal{F}[[P]] \equiv \{\langle f[[P]], \rho \rangle \mid \rho \in \mathcal{E}\}$.

The labelled transition semantics $T^\ell[[C]]$ of a command C in a program P is a set of transitions $\langle \sigma_1, A, \sigma_2 \rangle$ between a state σ_1 and its next states σ_2 by action A , satisfying the transition rule $\sigma_1 \xrightarrow{A} \sigma_2$.

$$\begin{aligned} T^\ell[{}^\ell \text{skip}] &\equiv \{\langle \ell, \rho \rangle \xrightarrow{{}^\ell \text{skip}} \langle f[{}^\ell \text{skip}], \rho \rangle \mid \rho \in \mathcal{E}\} \\ T^\ell[{}^\ell X := E] &\equiv \{\langle \ell, \rho \rangle \xrightarrow{{}^\ell X := E} \langle f[{}^\ell X := E], \rho[X \leftarrow v] \rangle \mid \rho \in \mathcal{E} \wedge v \in \mathbb{E}[[E]]\rho\} \\ T^\ell[\text{if } {}^\ell B \text{ then } C_1 \text{ else } C_2 \text{ }^{\ell'} \text{endif}] &\equiv T^\ell[[C_1]] \cup T^\ell[[C_2]] \cup \\ &\quad \{\langle \ell, \rho \rangle \xrightarrow{{}^\ell B} \langle i[[C_1]], \rho \rangle \mid \rho \in \mathcal{E} \wedge \text{true} \in \mathbb{B}[[B]]\rho\} \cup \\ &\quad \{\langle \ell', \rho \rangle \xrightarrow{{}^\ell \text{endif}} \langle f[\text{if } {}^\ell B \text{ then } C_1 \text{ else } C_2 \text{ }^{\ell'} \text{endif}], \rho \rangle \mid \rho \in \mathcal{E}\} \\ T^\ell[C_1; C_2] &\equiv T^\ell[[C_1]] \cup T^\ell[[C_2]] \\ T^\ell[\text{while } {}^\ell B \text{ do } C \text{ }^{\ell'} \text{done}] &\equiv \{\langle \ell, \rho \rangle \xrightarrow{{}^\ell \text{not } B} \langle \ell', \rho \rangle \mid \rho \in \mathcal{E} \wedge \text{false} \in \mathbb{B}[[B]]\rho\} \cup \\ &\quad \{\langle \ell, \rho \rangle \xrightarrow{{}^\ell B} \langle i[[C]], \rho \rangle \mid \rho \in \mathcal{E} \wedge \text{true} \in \mathbb{B}[[B]]\rho\} \cup T^\ell[[C]] \cup \\ &\quad \{\langle \ell', \rho \rangle \xrightarrow{{}^\ell \text{done}} \langle f[\text{while } {}^\ell B \text{ do } C \text{ }^{\ell'} \text{done}], \rho \rangle \mid \rho \in \mathcal{E}\} \end{aligned}$$

2.3 Concrete Domain

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

We define the partial trace semantics of a transition system as the set of all possible traces, denoted by Σ^* , recording the observation of an execution during a finite time, starting from an initial state and possibly reaching a final state.

$$\Sigma^* \in \wp(\Sigma \times \mathbb{A} \times \Sigma)$$

$$\Sigma^* = \{ \sigma_0 \xrightarrow{A_0} \dots \xrightarrow{A_{n-1}} \sigma_n \mid n \geq 1 \wedge \sigma_0 \in \mathcal{I} \wedge \forall i \in [0, n-1] : \sigma_i \xrightarrow{A_i} \sigma_{i+1} \in T^\ell \}$$

This set, with order relation “ \preceq ” and meet operator “ \wedge ”, forms the meet semi lattice $(\Sigma^*, \preceq, \wedge)$. Let $\pi_0, \pi_1 \in \Sigma^*$ be two partial traces, $\pi_0 \preceq \pi_1$ if and only if π_0 is a subtrace of π_1 and $\pi_0 \wedge \pi_1 = \pi$ such that $(\pi \preceq \pi_1) \wedge (\pi \preceq \pi_2)$ and $(\forall \pi' : (\pi' \preceq \pi_1) \wedge (\pi' \preceq \pi_2)). \pi' \preceq \pi$.

This partial trace semantics can be expressed also in fixpoint form.

$$\begin{aligned} \Sigma^* &= \text{lfp}^{\subseteq} F^t \text{ where} \\ F &\in \wp(\Sigma \times \mathbb{A} \times \Sigma) \rightarrow \wp(\Sigma \times \mathbb{A} \times \Sigma) \end{aligned}$$

Where

$$F(X) \equiv \{ \sigma \xrightarrow{A'} \sigma' \in T^\ell \mid \sigma \in \mathcal{I} \} \cup$$

$$\{ \sigma_0 \xrightarrow{A_0} \dots \xrightarrow{A_{n-2}} \sigma_{n-1} \xrightarrow{A_{n-1}} \sigma_n \mid \sigma_0 \xrightarrow{A_0} \dots \xrightarrow{A_{n-2}} \sigma_{n-1} \in X \wedge \sigma_{n-1} \xrightarrow{A_{n-1}} \sigma_n \in T^\ell \}$$

Let $\langle \wp(\Sigma^*), \sqsubseteq, \emptyset, \Sigma^*, \sqcap, \sqcup \rangle$ be a complete lattice of partial execution traces, where “ \sqsubseteq ”, “ \sqcap ” and “ \sqcup ” are defined as follows. Consider $\Pi^0, \Pi^1 \in \wp(\Sigma^*)$, $\Pi^0 \sqsubseteq \Pi^1$ if and only if $\forall \pi^0 \in \Pi^0. \exists \pi^1 \in \Pi^1$ such that $\pi^0 \preceq \pi^1$; $\Pi^0 \sqcup \Pi^1 = \{ \pi \in \Pi^0 \cup \Pi^1 \mid \forall \pi' \in \Pi^0 \cup \Pi^1, \pi' \preceq \pi \}$ and $\Pi^0 \sqcap \Pi^1 = \{ \pi \mid (\exists \pi^0 \in \Pi^0 \wedge \pi^1 \in \Pi^1). \pi = \pi^0 \wedge \pi^1 \wedge (\forall \pi' : (\pi' \preceq \pi^0) \wedge (\pi' \preceq \pi^1)). \pi' \preceq \pi \}$.

3 Abstract Domain

Our information leakage analysis combines a variable dependency analysis, based on propositional formulas, and variables’ value analysis, based on polyhedra, through the reduced product of the corresponding representation domains.

3.1 Variables Dependency Analysis

Propositional Formulas. Let $\mathbb{V}_p = \{ \bar{x}, \bar{y}, \bar{z}, \dots \}$ be a countably infinite set of propositional variables and let $FP(\mathbb{V}_p)$ be the set of finite subset of variables of \mathbb{V}_p . The set of propositional formulas constructed over the variables of \mathbb{V}_p and the logical connectives in $\Gamma \subseteq \{ \wedge, \vee, \rightarrow, \neg \}$ is denoted by $\Omega(\Gamma)$. For any $U \in FP(\mathbb{V}_p)$, $\Omega_U(\Gamma)$ consists of formulas using only the variables of U and the connectives of Γ .

A truth-assignment is a function $r : \mathbb{V}_p \rightarrow \{\text{true}, \text{false}\}$. Given a formula $f \in \Omega(\{\wedge, \vee, \rightarrow, \neg\})$, $r \models f$ means that r satisfies f , and $f_1 \models f_2$ is a shorthand for “ $r \models f_1$ implies $r \models f_2$ ”. $\Omega(\{\wedge, \vee, \rightarrow, \neg\})$ is ordered by $f_1 \sqsubseteq f_2$ if $f_2 \models f_1$. Two formulas f_1 and f_2 are logically equivalent, denoted $f_1 \equiv f_2$ if $f_1 \models f_2$ and $f_2 \models f_1$.

The unit assignment u is defined by $u(\bar{x}) = \text{true}$ for all $\bar{x} \in \mathbb{V}_p$. We define the set of positive formulas by: $Pos = \{f \in \Omega(\{\wedge, \vee, \rightarrow, \neg\}) \mid u \models f\}$, as in [5]. Some obvious examples: $T, \bar{x}_1 \in Pos$ and $F, \neg\bar{x}_1 \notin Pos$.

We can consider the propositional formula ϕ as a conjunction of subformulas $(\zeta_0 \wedge \dots \wedge \zeta_n)$. We denote the set of subformulas of ϕ as Sub_ϕ . Let ∇ be least upper bound operator on propositional formula, $\nabla\{\phi_0, \dots, \phi_n\} = \bigwedge\{Sub_{\phi_0}, \dots, Sub_{\phi_n}\}$. Therefore $(Pos, \sqsubseteq, \nabla)$ is a join semi lattice. Moreover, consider $\ominus : Pos \times Pos \rightarrow Pos$: a binary operator defined as subtraction between two propositional formulas: $\phi_0 \ominus \phi_1 = \bigwedge(Sub_{\phi_0} \setminus Sub_{\phi_1})$.

Abstract Domain. An abstract state $\sigma^\sharp \in \Sigma^\sharp \equiv \mathbb{L} \times Pos$ is a pair $\langle \ell, \phi \rangle$ which denotes the dependencies that occur among program variables, up to label $\ell \in \mathbb{L}$, expressed by the propositional formula $\phi \in Pos$. Given a pair $\sigma^\sharp = \langle \ell, \phi \rangle$, we define $l(\sigma^\sharp) = \ell$ and $r(\sigma^\sharp) = \phi$. Notice that the propositional variables are denoted by $\bar{\square}$. Let $BV(C)$ be the set of bound variables of command C .

$$\begin{aligned} BV(\ell skip) &= \{\emptyset\} \\ BV(\ell x := E) &= \{\bar{x}\} \\ BV(C_0; C_1) &= BV(C_0) \cup BV(C_1) \\ BV(\text{if } \ell B \text{ then } C_0 \text{ else } C_1 \text{ } \ell' \text{ endif}) &= BV(C_0) \cup BV(C_1) \\ BV(\text{while } \ell B \text{ do } C \text{ } \ell' \text{ done}) &= BV(C) \end{aligned}$$

The abstract transition semantics $\overline{T}^\ell[C]$ of a command C is a set of transition $\langle \sigma_1^\sharp, \sigma_2^\sharp \rangle$ between abstract states σ_1^\sharp and σ_2^\sharp . Similarly to the concrete domain we denote this transition by $\sigma_1^\sharp \rightarrow \sigma_2^\sharp$.

$$\begin{aligned} \overline{T}^\ell[\ell skip] &= \{\langle \ell, \phi \rangle \rightarrow \langle f[\ell skip], \phi \rangle\} \\ \overline{T}^\ell[\ell x := E] &= \{\langle \ell, \phi \rangle \rightarrow \langle f[\ell x := E], \phi' \rangle\} \\ \overline{T}^\ell[C_0; C_1] &= \overline{T}^\ell[C_0] \cup \overline{T}^\ell[C_1] \\ \overline{T}^\ell[\text{if } \ell B \text{ then } C_0 \text{ else } C_1 \text{ } \ell' \text{ endif}] &= \overline{T}^\ell[C_0] \cup \overline{T}^\ell[C_1] \cup \\ &\quad \{\langle \ell, \phi \rangle \rightarrow \langle i[C_0], \phi \rangle\} \cup \{\langle \ell, \phi \rangle \rightarrow \langle i[C_1], \phi \rangle\} \cup \\ &\quad \{\langle \ell', \phi \rangle \rightarrow \langle f[\text{if } \ell B \text{ then } C_0 \text{ else } C_1 \text{ } \ell' \text{ endif}], \phi''_{C_0} \rangle\} \\ &\quad \{\langle \ell', \phi \rangle \rightarrow \langle f[\text{if } \ell B \text{ then } C_0 \text{ else } C_1 \text{ } \ell' \text{ endif}], \phi''_{C_1} \rangle\} \\ \overline{T}^\ell[\text{while } \ell B \text{ do } C \text{ } \ell' \text{ done}] &= \overline{T}^\ell[C] \cup \{\langle \ell, \phi \rangle \rightarrow \langle i[C], \phi \rangle\} \cup \\ &\quad \{\langle \ell', \phi \rangle \rightarrow \langle f[\text{while } \ell B \text{ do } C \text{ } \ell' \text{ done}], \phi''' \rangle\} \end{aligned}$$

where:

$$\begin{aligned}\phi' &= \bigwedge \{\bar{y} \rightarrow \bar{x} \mid \bar{y} \in \mathbb{V}_p[[E]] \wedge \bar{y} \neq \bar{x}\} \wedge (\phi \ominus \bigwedge \{\bar{y} \rightarrow \bar{x} \mid \bar{y} \in \mathbb{V}_p \wedge \bar{x} \notin \mathbb{V}_p[[E]]\}) \\ \phi''_{C_0} &= \bigwedge \{y \rightarrow x \mid y \in \mathbb{V}_p[[B]] \wedge x \in BV(C_0) \wedge y \neq x\} \wedge \phi \\ \phi''_{C_1} &= \bigwedge \{y \rightarrow x \mid y \in \mathbb{V}_p[[B]] \wedge x \in BV(C_1) \wedge y \neq x\} \wedge \phi \\ \phi''' &= \bigwedge \{\bar{y} \rightarrow \bar{x} \mid \bar{y} \in \mathbb{V}_p[[B]] \wedge \bar{x} \in BV(C) \wedge \bar{y} \neq \bar{x}\} \wedge \phi\end{aligned}$$

Consider the set $\wp(\Sigma^\sharp)$, S_1 and S_2 , two sets of abstract state, such that

$$S_1 = \{\langle \ell_0^1, \phi_0^1 \rangle, \dots, \langle \ell_n^1, \phi_n^1 \rangle\} \quad S_2 = \{\langle \ell_0^2, \phi_0^2 \rangle, \dots, \langle \ell_m^2, \phi_m^2 \rangle\}$$

and $S_1 \sqsubseteq^\sharp S_2$ if and only if $n \leq m$, $\forall i \in [0, n]$, $\ell_i^1 = \ell_i^2$ and $\forall i \in [0, n]$, $\phi_i^1 \leq \phi_i^2$.

We can define a join and a meet operation on this set. Let $S_0, \dots, S_n \in \wp(\Sigma^\sharp)$ be sets of abstract states, the join operation “ \sqcup^\sharp ” is defined as:

$$\begin{aligned}\sqcup^\sharp \{S_0, \dots, S_n\} &= \bigcup (S_0, \dots, S_n) \\ &\cup \{\langle \ell, \phi \rangle \mid \phi = \nabla \{\phi' \mid \langle \ell, \phi' \rangle \in \bigcup (S_0, \dots, S_n)\}\} \\ &\setminus \{\langle \ell, \phi \rangle \in \bigcup (S_0, \dots, S_n) \mid \exists \langle \ell, \phi' \rangle \in \bigcup (S_0, \dots, S_n) \wedge \phi \neq \phi'\}\end{aligned}$$

and the meet operation “ \sqcap^\sharp ”:

$$\begin{aligned}\sqcap^\sharp \{S_0, \dots, S_n\} &= \{\langle \ell, \phi \rangle \in S' \mid S' \in \{S_0, \dots, S_n\} \wedge \\ &\quad \forall i \in [0, n]. \exists \langle \ell, \phi'_i \rangle \in S_i \wedge \phi \leq \phi'_i\}\end{aligned}$$

Therefore, $\langle \wp(\Sigma^\sharp), \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcap^\sharp, \sqcup^\sharp \rangle$ is a complete lattice.

Let $\mathcal{I}^\sharp[[P]] = \{\langle i[[P]], \phi \rangle \mid \phi \in Pos\}$ be the set of possible initial abstract state of program P . We define the abstract semantics as the set of all finite sets of abstract states, denoted by $\Sigma^{*\sharp}$, which can occur during one or more execution, in a finite time. For each element $\mathbb{S} \in \Sigma^{*\sharp}$ we can denote by \mathbb{S}^\perp the set of terminal states, defined as $\mathbb{S}^\perp = \{\sigma_0^\sharp \mid \nexists \sigma_1^\sharp \in \mathbb{S}. \sigma_0^\sharp \rightarrow \sigma_1^\sharp \in \overline{T}^\ell\}$ and by $\ell(\mathbb{S})$ all labels of the set \mathbb{S} . Letting $\mathbb{S}_{\sigma_0^\sharp, \sigma_n^\sharp}$ denote a set of states, called abstract sequence, that contains a starting state σ_0^\sharp and an ending state σ_n^\sharp such that $\forall i \in [0, n-1]$, $\sigma_i^\sharp \rightarrow \sigma_{i+1}^\sharp \in \overline{T}^\ell$. Notice that $\mathbb{S}_{\sigma_0^\sharp, \sigma_n^\sharp}^\perp = \{\sigma_n^\sharp\}$.

We express the abstract semantics in fixpoint form.

$$\begin{aligned}\Sigma^{*\sharp} &= lfp^\square F^\sharp \text{ where} \\ F^\sharp &\in \Sigma^{*\sharp} \rightarrow \Sigma^{*\sharp}\end{aligned}$$

where

$$\begin{aligned}F^\sharp(X) &\equiv \{\sigma^\sharp \mid \sigma^\sharp \in \mathcal{I}^\sharp\} \cup \\ &\quad \{\mathbb{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \mid n \geq 1 \wedge \sigma_0^\sharp \in \mathcal{I}^\sharp \wedge \mathbb{S}_{\sigma_0^\sharp, \sigma_{n-1}^\sharp} \in X \wedge \sigma_{n-1}^\sharp \rightarrow \sigma_n^\sharp \in \overline{T}^\ell\} \cup \\ &\quad \{\sqcup^\sharp \{\mathbb{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \mid \mathbb{S}_{\sigma_0^\sharp, \sigma_n^\sharp} \in X\}\}\end{aligned}$$

Let $\langle \Sigma^{*\sharp}, \sqsubseteq^\sharp, \emptyset, \Sigma^\sharp, \sqcap^\sharp, \sqcup^\sharp \rangle$ be a lattice of abstract state sets, our abstract domain.

To simplify the definition of Galois connection we present another domain, isomorphic to the concrete domain. Let $\sigma^\diamond \in \Sigma^\diamond \equiv \mathbb{L} \times \mathbb{A}$ be a pair $\langle \ell, A \rangle$

where A is the action which occur at program label $\ell \in \mathbb{L}$. Consider the set $\Sigma^{*\diamond}$ which contains all the possible sequence of σ^\diamond that can occur during a finite computation, and the lattice $\langle \wp(\Sigma^{*\diamond}), \sqsubseteq^\diamond, \emptyset, \Sigma^{*\diamond}, \cap, \cup \rangle$. We define for $\Pi_0^\diamond, \Pi_1^\diamond \in \wp(\Sigma^{*\diamond})$, $\Pi_0^\diamond \sqsubseteq^\diamond \Pi_1^\diamond$ if and only if for each $\pi_0^\diamond \in \Pi_0^\diamond$ exists $\pi_1^\diamond \in \Pi_1^\diamond$ such that $\pi_0^\diamond \preceq^\diamond \pi_1^\diamond$, while $\pi_0^\diamond \preceq^\diamond \pi_1^\diamond$ if and only if π_0^\diamond is a subsequence of π_1^\diamond . Moreover, we denote by $\pi^{\diamond^{-1}}$ the last state of the sequence.

We can express the relation between $\wp(\Sigma^*)$ and $\wp(\Sigma^{*\diamond})$ by two functions: the abstraction function, $\alpha^\diamond \in \wp(\Sigma^*) \rightarrow \wp(\Sigma^{*\diamond})$, and the concretization function, $\gamma^\diamond \in \wp(\Sigma^{*\diamond}) \rightarrow \wp(\Sigma^*)$.

Let $X = \{\pi_0, \dots, \pi_n\} \in \wp(\Sigma^*)$ be a set of partial trace and let $Y = \{\pi_0^\diamond, \dots, \pi_n^\diamond\} \in \wp(\Sigma^{*\diamond})$ be a set of sequences of σ^\diamond .

$$\begin{aligned} \alpha^\diamond(X) &\equiv \{ \langle \ell_0, A_0 \rangle \rightarrow \dots \rightarrow \langle \ell_m, A_m \rangle \mid \sigma_0 \xrightarrow{\ell_0 A_0} \dots \xrightarrow{\ell_m A_m} \sigma_{m+1} \in X \} \\ \gamma^\diamond(Y) &\equiv \{ \pi \in \wp(\Sigma^*) \mid \alpha^\diamond(\{\pi\}) \in Y \} \end{aligned}$$

Notice that $(\gamma^\diamond, \wp(\Sigma^*), \wp(\Sigma^{*\diamond}), \alpha^\diamond)$ is an isomorphism: in fact it's simple to prove that $\gamma^\diamond \circ \alpha^\diamond = \alpha^\diamond \circ \gamma^\diamond = id$, where id is the identity function.

Then, we can define the relation between $\wp(\Sigma^{*\diamond})$ and $\Sigma^{*\sharp}$ by the abstraction function α^\sharp and γ^\sharp . Let $X \in \wp(\Sigma^{*\diamond})$ be a set of sequences of σ^\diamond , $\alpha^\sharp : \wp(\Sigma^{*\diamond}) \rightarrow \Sigma^{*\sharp}$ is defined as $\alpha^\sharp(X) = \sqcup^\sharp \{ \theta(\pi^\diamond) \mid \pi^\diamond \in X \}$, where $\theta : \Sigma^{*\diamond} \rightarrow \Sigma^{*\sharp}$ is defined ad follows.

$$\begin{aligned} \theta(X) &= \{ \langle \ell, \phi \rangle \mid \forall \pi \in X. \forall \pi' = \langle \ell_0, A_0 \rangle \rightarrow \langle \ell_m, A_m \rangle \preceq^\diamond \pi : \\ &\quad m \geq 0 \wedge \ell = \ell_m \wedge \phi = f_0 \wedge \dots \wedge f_n \} \end{aligned}$$

such that:

- $(\forall \langle \ell, x := E \rangle \in \pi' : \forall \langle \ell', x := E' \rangle \in \pi'. \ell' \leq \ell, \exists f_i = \bar{y} \rightarrow \bar{x} : \bar{y} \in \mathbb{V}_p \llbracket E \rrbracket$
- $\forall (\langle \ell_i, B \rangle \rightarrow \dots \rightarrow \langle \ell_j, endif \rangle) \vee (\langle \ell_i, not B \rangle \rightarrow \dots \rightarrow \langle \ell_j, endif \rangle) \preceq^\diamond \pi^\diamond$ which represents an *if* statement and $\forall \langle \ell_k, x := E_k \rangle : i < k < j$ exists $f_h = \bar{y} \rightarrow \bar{x}$ such that $\bar{y} \in \mathbb{V}_p \llbracket B \rrbracket$.
- $\forall (\langle \ell_i, B \rangle \rightarrow \dots \rightarrow \langle \ell_j, done \rangle) \vee (\langle \ell_i, not B \rangle \rightarrow \dots \rightarrow \langle \ell_j, done \rangle) \preceq^\diamond \pi^\diamond$ which represents an *while* statement and $\forall \langle \ell_k, x := E_k \rangle : i < k < j$ exists $f_h = \bar{y} \rightarrow \bar{x}$ such that $\bar{y} \in \mathbb{V}_p \llbracket B \rrbracket$.

Notice that $\langle \ell_i, B \rangle \rightarrow \dots \rightarrow \langle \ell_j, endif \rangle$ (or $\langle \ell_i, not B \rangle \rightarrow \dots \rightarrow \langle \ell_j, endif \rangle$) represents an *if* statement if and only if $\forall (\langle \ell_p, B \rangle \vee \langle \ell_p, not B \rangle) : i < p < j. \exists (\langle \ell_q, endif \rangle \vee \langle \ell_q, done \rangle) : p < q < j$ and $\forall (\langle \ell_q, endif \rangle \vee \langle \ell_q, done \rangle) : i < q < j. \exists \langle \ell_p B \vee \langle \ell_p, not B \rangle) : i < p < q$. Similarly for *while* statement.

On the other hand, the concretization function $\gamma^\sharp : \Sigma^{*\sharp} \rightarrow \wp(\Sigma^{*\diamond})$ is defined as follows. Let $Y \in \Sigma^{*\sharp}$:

$$\gamma^\sharp(Y) = \{ \pi^\diamond \in \Sigma^{*\diamond} \mid \theta(\pi^\diamond) \sqsubseteq^\sharp Y \wedge l(\pi^{\diamond^{-1}}) \in l(Y^{-1}) \}$$

It's simple to prove that γ^\sharp and α^\sharp are monotone, $\gamma^\sharp \circ \alpha^\sharp$ is extensive, $\alpha^\sharp \circ \gamma^\sharp$ is equivalent to the identity and that $(\gamma^\sharp, \wp(\Sigma^{*\diamond}), \Sigma^{*\sharp}, \alpha^\sharp)$ is a Galois insertion.

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

By property of function composition we can assert that $(\gamma, \wp(\Sigma^*), \Sigma^{*\sharp}, \alpha)$ is a Galois insertion.

3.2 Polyhedra Analysis

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. In the seminal work [7], P. Cousot and N. Halbwachs applied the theory of abstract interpretation to the static determination of linear equality and inequality relations among program variables and introduced the use of convex polyhedra as a domain of descriptions to solve a number of important data-flow analysis problems.

There are many works in literature on the use of the polyhedra domain and relative Galois connection and on its implementations [1,12]. Therefore we can omit a comprehensive presentation of polyhedra analysis and provide only some basic notions.

For $n > 0$ we denote by $\mathbf{v} = (v_0, \dots, v_{n-1}) \in \mathbb{R}^n$ an 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 $\mathbf{0} \in \mathbb{R}^n$ has all components equal to zero. Let \mathbf{x} be a n -tuple of distinct variable. Then $\beta = (\mathbf{a} \cdot \mathbf{x} \bowtie b)$ denotes a linear equality and inequality constraint, for each vector $\mathbf{a} \in \mathbb{R}^n$, where $\mathbf{a} \neq \mathbf{0}$, each scalar $b \in \mathbb{R}$ 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 $\mathcal{P} = con(\mathcal{C})$ of a finite set of linear inequality constraints \mathcal{C} . The set of all polyhedra on the vector space \mathbb{R}^n is denoted as \mathbb{P}_n . Let $\langle \mathbb{P}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap \rangle$ 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, returning the greatest polyhedron smaller than or equal to the two arguments, correspond to set-intersection and “ \uplus ” is the binary join operation and return the least polyhedron greater than or equal to the two arguments, called convex polyhedral hull. Moreover let $G_{\wp(\Sigma^*), \mathbb{P}_n} = (\gamma_{\mathbb{P}_n, \wp(\Sigma^*)}, \wp(\Sigma^*), \mathbb{P}_n, \alpha_{\wp(\Sigma^*), \mathbb{P}_n})$ be a Galois connection between the concrete domain $\wp(\Sigma^*)$ and abstract domain \mathbb{P}_n .

3.3 Reduced Product

We combine the abstract domains $\langle \Sigma^{*\sharp}, \sqsubseteq^\sharp, \emptyset, \Sigma^{*\sharp}, \sqcap^\sharp, \sqcup^\sharp \rangle$ and $\langle \mathbb{P}_n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap \rangle$ through a reduced product operator [6].

Let $G_{\wp(\Sigma^*), \Sigma^{*\sharp}}$ and $G_{\wp(\Sigma^*), \mathbb{P}_n}$ be Galois connection and let $\varrho : \Sigma^{*\sharp} \times \mathbb{P}_n \rightarrow \Sigma^{*\sharp} \times \mathbb{P}_n$ be a reduction operator defined as follows: let $X \in \Sigma^{*\sharp}$ be a set of partial traces and let $\mathcal{P} \in \mathbb{P}_n$ be a polyhedra.

$$\varrho(\langle X, \mathcal{P} \rangle) = \langle X', \mathcal{P} \rangle$$

such that

$$X' = \{ \sigma_{new}^\sharp \mid \forall \sigma^\sharp \in X. l(\sigma_{new}^\sharp) = l(\sigma^\sharp) \wedge \wedge r(\sigma_{new}^\sharp) = (r(\sigma^\sharp) \ominus \{x \rightarrow y \mid y = z \in \mathcal{P}, z \in \mathbb{V}_p \cup \mathbb{Z} \wedge z \neq x\}) \}$$

Then, the reduced product D^\sharp is defined as follows:

$$D^\sharp = \{ \varrho(\langle X, \mathcal{P} \rangle) \mid X \in \Sigma^{*\sharp}, \mathcal{P} \in \mathbb{P}_n \}$$

Consider $X_0, X_1 \in \Sigma^{*\sharp}, \mathcal{P}_0, \mathcal{P}_1 \in \mathbb{P}_n$ and $\langle X_0, \mathcal{P}_0 \rangle, \langle X_1, \mathcal{P}_1 \rangle \in D^\sharp: \langle X_0, \mathcal{P}_0 \rangle \sqsubseteq^\sharp \langle X_1, \mathcal{P}_1 \rangle$ if and only if $X_0 \sqsubseteq^\sharp X_1$ and $\mathcal{P}_0 \subseteq \mathcal{P}_1$. Let $\sqcup^\sharp : D^\sharp \rightarrow D^\sharp$ and $\sqcap^\sharp : D^\sharp \rightarrow D^\sharp$ be the least upper bound and greatest lower bound operator, respectively, defined as $\langle X_0, \mathcal{P}_0 \rangle \sqcup^\sharp \langle X_1, \mathcal{P}_1 \rangle = \langle X_0 \sqcup^\sharp X_1, \mathcal{P}_0 \uplus \mathcal{P}_1 \rangle$ and $\langle X_0, \mathcal{P}_0 \rangle \sqcap^\sharp \langle X_1, \mathcal{P}_1 \rangle = \langle X_0 \sqcap^\sharp X_1, \mathcal{P}_0 \cap \mathcal{P}_1 \rangle$.

Therefore $\langle D^\sharp, \sqsubseteq^\sharp, \emptyset, \varrho(\langle \Sigma^{*\sharp}, \mathbb{R}^n \rangle), \sqcup^\sharp, \sqcap^\sharp \rangle$ is a complete lattice.

The reduce operator showed above is aimed at excluding pointless dependencies for all variables which have the same value during the execution, without the loss of purposeful relations (by the condition “ $x \neq z$ ”). In order to better understand the improvements yielded by the combination of the two domains consider the following example.

Example 2.

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

For the sake of simplicity, we show a partial representations through propositional formula and polyhedra of the variables dependency. In particular we take in account the labels 4, 5, 8, 10, 12 and 14.

	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:

$$\begin{array}{l|l}
4 & \mathbf{T} \\
5 & p \rightarrow \text{sum} \\
8 & p \rightarrow \text{sum} \\
10 & p \rightarrow \text{sum} \\
12 & p \rightarrow \text{sum} \\
14 & (p \rightarrow \text{sum}) \wedge (i \rightarrow n) \wedge (k \rightarrow n)
\end{array}$$

By using the reduce operator we simplified the propositional formulas, removing some implication which could in fact generate false alarms when using the direct product of the domains instead of the reduced product.

4 Analysis

An information flow analysis can be carried out by considering different attacker abilities. In this paper, we consider 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. Consider that the attacker, in both cases, knows the source code of the program.

Let $\mathcal{Y}_P : \mathbb{V} \rightarrow \{L, H\}$ be a function which assigns to each variable of program P a security class: public (L) or private (H). We say that program P is secure if and only if it does not contain any information leakage with respect to the function \mathcal{Y}_P , i.e. there is no information that moves from private to public variables.

Let $\overline{\mathcal{Y}}_P : \mathbb{V}_p \rightarrow \{\mathbf{T}, \mathbf{F}\}$ be the truth-assignment function associated with \mathcal{Y}_P . $\overline{\mathcal{Y}}_P(\overline{x})$ assigns to \overline{x} the value \mathbf{T} or \mathbf{F} if the security class, assigned by \mathcal{Y}_P , is H or L respectively.

The aim of this analysis is to formally verify if the program is secure, therefore we look for all subsets of the prefix partial trace that do not have any information leakage. For the first case, in which the attacker can read public variables only at the beginning and at the end of the computation, the property is $\chi^1 = \{\mathbb{S} \in \Sigma^{*\sharp} \mid \overline{\mathcal{Y}}_p \models r(\mathbb{S}^\dagger)\}$ which is all abstract sequences in $\Sigma^{*\sharp}$ such that the propositional formula of the last abstract state is satisfied by truth-assignment function $\overline{\mathcal{Y}}_p$. Whereas in the second case, when the attacker can read public variables at each step of the computation, the property is $\chi^2 = \{\mathbb{S} \in \Sigma^{*\sharp} \mid \forall \sigma^\sharp \in \mathbb{S}. \overline{\mathcal{Y}}_p \models r(\sigma^\sharp)\}$ the set of all abstract sequences in $\Sigma^{*\sharp}$ which contains all the abstract states that are satisfied by the truth-assignment function $\overline{\mathcal{Y}}_p$.

4.1 Complexity

In order to evaluate the efficiency of our analysis, we have to consider the two main components: *Pos* formulas and polyhedra.

Variables dependency analysis through propositional formulas involves two different aspects: on the one hand, the logical equivalence of two boolean expressions, a co-NP-complete problem, and on the other hand the fact that the complexity of *Pos* domain is bounded because we work only with the variables appearing in the program, whose number is, in the practice, reasonably small [11]. Moreover, it is possible to reduce the complexity using the ordered binary

decision diagrams (BDDs) to provide a compact representation of many boolean functions and by using algorithms based on that.

About polyhedra analysis, the complexity is well and completely treated in many works [1] and heavily depends on its implementation.

For example many implementation, 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 expand (in amortized constant time) ensuring the best use of available memory. Comparing the efficiency of the polyhedra libraries is not simple, the pay-off depends on the targeted applications: in [1] the authors presented many test results about it.

In this paper we considered the polyhedra analysis, but, as already observed, the modular construction allows to tune efficiency and accuracy changing the domain which represents the relations among variables' values.

For instance, we can use the analysis proposed by Karr in [13]: in this way we may get a loss of precision, as this domain represents only linear combination of the variables, but we achieve an improvement of the computational cost of the analysis (it is polynomial).

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

5 Conclusions

In this paper, we presented an information flow analysis through abstract interpretation based on a new domain that combines variable dependency analysis, based on the reduced product of a propositional formulas domain, and a polyhedra domain. The techniques used in our analysis are frequently applied in the context of generic code analysis; the main contribution of our work consists in combining these techniques in a novel way to solve a specific security problem: the detection of information leakages.

Moreover, the structure of our analysis allow easily some generalization, e.g. for multi-level security policies or for other kind of analysis (for example, knowing which variables depends on others, regardless of the security classes).

Acknowledgments

Work partially supported by RAS project "TESLA - Tecniche di enforcement per la sicurezza dei linguaggi e delle applicazioni".

References

1. 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. *Science of Computer Programming* 72(1–2), 3–21 (2008)

2. Bodei, C., Degano, P., Nielson, F., Riis Nielson, H.: Static analysis for secrecy and non-interference in networks of processes. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 27–41. Springer, Heidelberg (2001)
3. Braghin, C., Cortesi, A., Focardi, R.: Information flow security in boundary ambi-ents. *Inf. Comput.* 206(2-4), 460–489 (2008)
4. Centenaro, M., Focardi, R., Luccio, F.L., Steel, G.: Type-based analysis of pin processing apis. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 53–68. Springer, Heidelberg (2009)
5. Cortesi, A., File, G., Winsborough, W.: Optimal groundness analysis using propositional logic. *The Journal of Logic Programming* 27(2), 137–167 (1996)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 269–282. ACM Press, New York (1979)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 84–97. ACM Press, New York (1978)
8. Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* 19(5), 236–243 (1976)
9. 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)
10. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, vol. 0, p. 11 (1982)
11. Van Hentenryck, P., Cortesi, A., Le Charlier, B.: Evaluation of the domain prop. *The Journal of Logic Programming* 23(3), 237–278 (1995)
12. Jeannet, B.: Convex Polyhedra Library, release 1.1.3c edn., Documentation of the “New Polka” library (March 2002), <http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html>
13. Karr, M.: Affine relationships among variables of a program. *Acta Inf.* 6, 133–151 (1976)
14. Smith, G.: Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection. Advances in Information Security, vol. 27, pp. 291–307. Springer, Heidelberg (2007)
15. Tolstrup, T.K., Nielson, F., Nielson, H.R.: Information flow analysis for vhd. In: PaCT, pp. 79–98 (2005)
16. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. *J. Comput. Secur.* 4(2-3), 167–187 (1996)